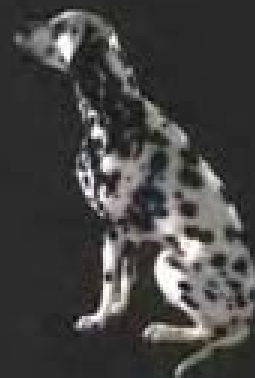




万水计算机技术实用大全系列



[美] Konrad King 著

杜大鹏 龚小平 史艳辉 纪广民 等译

杜国梁 审校

SQL

编程实用大全(精华版)

SQL Tips and Techniques



中国水利水电出版社
www.waterpub.com.cn

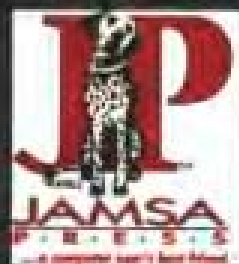
SQL 编程实用大全 (精华版)

一页一页地阅读,从中找出自己所需的技巧!

拥有这本讲述了从表的组成到执行批处理更新的几乎有关 SQL 的一切方面的参考书是绝对必要的!本书从为第一次使用 SQL 编程的人员提供常用技巧开始,然后循序渐进地提高读者使用 SQL 的技艺。等到读完全部技巧时,读者就已经学完了有关 SQL 的几乎所有方面的知识。或者,读者只是将本书作为一本内容广泛的参考书,用来快速地得到某些特定的 SQL 问题的答案。不管读者如何使用本书,这本书都是读者书架上必备的一本宝贵的 SQL 参考书。

作者简介

Konrad King 是一位居住在内华达州拉斯维加斯的程序员和数据库管理员。这位美国空军学院的毕业生在实时编程、网络协议、基于 Web 的应用程序,以及数据库管理方面有较强的造诣。



ONWORD PRESS
THOMSON LEARNING

ISBN 7-5084-2893-5



9 787508 428932 >

责任编辑:宋俊娥

封面设计:王 茹

本书内容涵盖 SQL 的所有方面

- ◆ 使用视图增加安全性并简化多表查询
- ◆ 用主键和外键约束维护数据的完整性
- ◆ 使用存储过程和触发器使任务的完成自动化
- ◆ 将多个 Update、Delete 和 Insert 语句看作是单个事务处理
- ◆ 使用触发器发送邮件消息
- ◆ 通过控制对数据库对象的访问权限来实现数据库的安全性
- ◆ 创建索引加快数据库查询的执行速度
- ◆ 编写处理数据库中数据的 Visual Basic 应用程序
- ◆ 建立动态 Web 页面
- ◆ 使用查询生成记录集和可更新的游标
- ◆ 使用嵌套查询来处理多个表
- ◆ 实现备份策略并使用事务处理日志恢复数据库
- ◆ 一次改变多行中的值
- ◆ 执行动态 SQL 语句
- ◆ 组合多个查询的结果

还有比这更多的内容等待读者去探索!



清华大学出版社

地址:北京清华大学学研大厦A座

邮编:100084

电话:(010)62770175

http://www.tup.com.cn

01-0000000000000000

01-0000000000000000

01-0000000000000000

万水计算机技术实用大全系列

SQL 编程实用大全

(精华版)

[美] Konrad King 著

杜大鹏 龚小平 等译
史艳辉 纪广民

杜国梁 审校

中国水利水电出版社

内 容 提 要

本书以技巧形式讲述了有关 SQL 的各个方面。作者以其丰富的 SQL 数据库工作经验,向读者介绍了从数据库概念到数据库理论,从 SQL 标准到各种 SQL 数据库产品,从数据库的编程方法到具体的语句句法,从分布式计算到 Internet 应用等的有关知识。本书与众多的介绍某种软件的使用与操作方法的书籍不同,以 SQL-89 和 SQL-92 两种规范为基础,着重介绍 SQL 数据库各种产品的共同的基础知识和编程方法,在涉及具体软件时,重点介绍了 MS-SQL Server 的实现方式,但同时比较与其他软件的异同。本书是 SQL 知识的大全。读者既可作为学习 SQL 知识的教科书,循序渐进地学习各方面的知识,也可作为手边的参考资料,在学习和工作中遇到问题时随时查阅。书中的大量示例代码具有很高的实用性,读者略加修改就可以用在自己的编程实践中。

本书可以作为从事数据库软件开发、Internet 网站设计以及电子商务等技术的中高级程序员的参考书。

Authorized translation from English Language Edition published by Premier Press, Inc. Original copyright © 2002 by Premier Press, SQL Tips & Techniques. Translation by China WaterPower Press, 2003.

北京市版权局著作权合同登记号:图字 01-2002-3539

图书在版编目(CIP)数据

SQL 编程实用大全:精华版(美)金(King, K.)著;杜大鹏等译. —北京:中国水利水电出版社, 2005

(万水计算机技术实用大全系列)

书名原文:SQL Tips & Techniques

ISBN 7-5084-2893-5

I. S… II. ①金…②杜… III. 关系数据库—数据库管理系统, SQL—程序设计 IV. TP311.138

中国版本图书馆 CIP 数据核字(2005)第 046633 号

书 名	SQL 编程实用大全(精华版)
作 者	[美] Konrad King 著
译 者	杜大鹏 龚小平 史艳辉 纪广民 等译
审 校	杜国梁
出版 发行	中国水利水电出版社(北京市三里河路 6 号 100044) 网址: www.waterpub.com.cn E-mail: mchannel@263.net (万水) sales@waterpub.com.cn 电话: (010) 63202266 (总机)、68331835 (营销中心)、82562819 (万水)
经 售	全国各地新华书店和相关出版物销售网点
排 版	北京万水电子信息有限公司
印 刷	北京蓝空印刷厂
规 格	787mm×1092mm 16 开本 40.75 印张 917 千字
版 次	2005 年 5 月第 1 版 2005 年 5 月第 1 次印刷
印 数	0001—4000 册
定 价	65.00 元

凡购买我社图书,如有缺页、倒页、脱页者,本社营销中心负责调换
版权所有·侵权必究

精华版序

《万水计算机技术实用大全系列》自出版以来，受到了广大读者的欢迎和好评。在第一版售罄以后，仍有不少读者期望购买此套丛书，更有读者期望收藏此套丛书。为此我们征求了广大读者的意见，在第一版的基础上对整套丛书进行了改编，去掉了一些空洞的技巧，合并了罗嗦的章节段落，删除了陈旧的内容，使其更加符合中国读者的习惯，更加便于查阅。同时，从读者的角度考虑，我们在版面编排上略加压缩，在第一版的基础上缩减了图书的页面篇幅，让读者能以更低的价格获得更好的图书。

当今的计算机技术日新月异，软件的新版本不断推出，但其中除了技术本身的发展和创新之外，还存在着软件供应商的商业行为。本套丛书囊括了 Visual Basic、SQL、Visual C++ 和 C/C++/C# 软件的各种实用技巧，是开发人员案头必备的一套参考丛书。当然，有人会说，部分软件已经有了新版本。从一方面考虑，众多读者还在使用原有版本，这套丛书非常适合于他们；而另一方面，即使读者采用了新版本的软件，但由于这些新旧版本软件所基于的底层技术是一样的，这套丛书同样也非常适合于他们。其实，读者的需求已经告诉我们，主流计算机软件的基本技术短期内不会改变，那么运用它们的基本技巧也不会随时间的流逝而淹没。这正是我们重新改编出版这套《万水计算机技术实用大全系列》精华版的初衷。我们相信，读者会喜欢这套丛书，也会从本套丛书中受益匪浅。

如果您是一名开发人员，那么一定很明白，软件需求千变万化，其代码实现也千变万化。在开发过程，经常会碰到为了解决某个问题而绞尽脑汁。您会疯狂地查阅网上资料或者图书资料，希望能够拓展思路或是获得具体实现方法，这时您多么渴望能有一本收集了各种实用技巧的图书！本套丛书的写作初衷就在于此，它融入了作者长期的开发经验，收集了相关软件的各种实用开发技巧。本套丛书最大的特点就是按软件功能给出一条条实用技巧，技巧之间相互引用，这样方便读者查阅。举例来说，如果您对 SQL 中的 BIT 数据类型不太了解，可以查阅《SQL 编程实用大全（精华版）》一书的目录，从第 1 章“理解 SQL 基本知识并创建数据库文件”中找到“技巧 16 理解 SQL 的 BIT 数据类型”，阅读该技巧即可获得相关知识。软件开发人员的工作时间往往都很长，如果有了这么一套丛书，当您冥思苦想某种实现方法时，不妨查阅一下本书，也许您可以多腾出几晚上的时间来喝咖啡，也许您能更多地赢得老板的赏识，让工作变得更轻松。

丛书精华版在改编过程中力求突出精华二字。首先是对上一版的文字进行了改编，使内容更简洁，更符合中文阅读习惯；其次删除了一些空洞的技巧、过于基础的内容、陈旧的内容、罕见的內容，或者是在新版本软件中已经不再采用的方法；最后丛书精华版在版面上进行了压缩，在较短的篇幅中容纳更多的内容，更适于作为参考书。本套丛书精华版适合于 Visual Basic、SQL、Visual C++ 和 C/C++/C# 开发人员作为参考书阅读和查阅，是开发人员解决实际问题 and 拓宽思路不可多得的一套丛书。

最后，感谢参与本套丛书改编的几位老师和开发人员，他们是互动出版网的姚先勤、哈尔滨理工大学的徐继伟、华北电力大学的燕士跃、北京英宇世纪信息技术有限公司的李秋霞。其中部分内容融入了他们的开发实践，在此谨表感谢！

译 者 序

与十年前甚至五年前相比，现在计算机应用可谓随处可见。人们去超市购物，有收款机收款；去银行存款或取款，由计算机处理，通存通兑，十分方便；现在到邮局办理业务，也会发现几乎所有业务都变为由计算机处理了；上网现在已不是什么神秘的事情，收发电子邮件，网上购物，在网上买卖股票和基金。所有这一切应用的背后，除了网络硬件、操作系统软件的支持之外，还有一个“幕后英雄”，那就是数据库和数据库管理系统。甚至看似与计算机无关的应用，比如我们日常生活中离不开的电话，也是数据库和数据库管理系统在后台为我们“日夜辛劳”。离开这种软件，以上应用都是不可能的。SQL (Structured Query Language, 结构化查询语言) 正是数据库和数据库管理系统的人机交互接口或界面，通过这种语言，人们只要告诉数据库管理系统做什么并获得所做事物的结果，至于怎么做，由数据库管理系统负责。虽然一般使用计算机的人不需要了解 SQL，但作为计算机软件开发人员，特别是对网络应用、电子商务和企业管理等方面的软件开发人员来说，掌握这一技术就是从事该种工作的必备条件之一。本书以一条一条技巧的形式向读者讲授了 SQL 知识的方方面面，为初涉这一领域的人起到了领路的作用，对于此行业资深人员或许也有一定的参考价值。

本书形式新颖，内容全面，叙述深入浅出，是一本不可多得的有关 SQL 知识的参考书。

在本书翻译过程中，正值世界杯赛事如火如荼地进行，译者大多数是球迷，这样的机会当然不能错过。由于观球影响了进度，只好多开几个夜车加以弥补。等到翻译进度达到尾声时，又赶上了闷热难熬的天气。译者们克服了这一切困难，现在终于将这部书稿呈现在读者面前了。面对像砖头一样厚的一部书稿摆在面前，除了无比欣慰之外，也期待着读者评头品足。

杜人鹏、任建畅、梁兴泉、龚小平、史艳辉、纪广民等参加了翻译工作，其中，杜人鹏翻译了第 1 章~第 7 章，任建畅翻译了第 8 章~第 12 章，梁兴泉翻译了第 13 章~第 15 章，龚小平翻译了第 16 章~第 20 章，史艳辉翻译了第 21 章~第 23 章，纪广民翻译了第 24 章和第 25 章。全书由杜国梁审校并统稿。参加本书其他工作（录入、打印、校对等）的有魏全宇、梁国珍、杜墨、梁大伟、马相生、刘发来、董明、迟春和杨天华等。没有这些人的共同努力，要想在如此短的时间内完成这样大部头的书稿翻译工作是不可能的，在此对他们为本书所做出的贡献表示感谢。

译 者

2002 年 8 月

作者简介

Konrad King 在中学的低年级时就在 Lancaster Community College (兰卡斯特社区学院) 参加了夜校的计算机课程 (学习 COBOL 语言), 从那时起就开始编写程序并使用计算机。作为一名优秀生从 Mojave High School (莫扎夫高级中学) 毕业之后, Konrad 就读于 U.S. Air Force Academy (美国空军学院) 并获得计算机专业科学学士学位。除了获得不少学术奖学金之外, Konrad 还以全班第三名的成绩从空军学院毕业并以计算机科学主修的顶级成绩获得 Eagle and Fledglings 奖学金。

1984 年, Konrad 作为服役军官加入美国空军, 以 Data General MV 微机系列的系统管理员身份服役四年。在这一职位上, Konrad 经常在硬件和软件维护协议上与经销商打交道, 经手过几百万美元的设备采购, 实施过大量的备份策略, 管理并维护所有的计算机系统以及应用程序, 而且还用 FORTRAN 语言编写过实时数据采集程序。

1988 年离开美国空军之后, Konrad 在拉斯维加斯 (Las Vegas) 开始了自己的咨询业务。其主要精力放在开发企业数据库系统上, 允许客户使用一套自定义的、用户友好的应用程序来管理其业务的各个方面。到这时, Konrad 使用 Dbase 和 Dataflex 关系数据库系统编写了界面程序, 而最近使用 Visual C++ 和 Visual Basic 为 Microsoft、Oracle 和 Sybase SQL DBMS 产品编写了更多的程序。除了通过编写无数行的捕获数据生成关键的管理和生产报告的代码而不断提高编程技艺之外, Konrad 在收集了大量的 Windows NT/2000/XP 联网、Novell 网络和 SQL DBMS 安装、备份和性能方面的知识。Konrad 曾经在软件和硬件级别上, 在大型主机、微机、PC 机上工作过, 从头开始建立过 PC 和基于 PC 的网络。

Konrad 的最新努力包括设计并实施几个允许客户改善顾客关系并通过电子商务来扩展其业务的 Web 站点。使用 ASP Scripts、Java 小程序和 ActiveX 对象的组合, Konrad 的 Web 站点通过建立跨 Internet 的双向通信允许顾客与服务人员联系, 在线查看其基于 SQL 服务器的账户信息, 在受保护的 Web 服务器上使用信用卡购买物品。

在业余时间里, Konrad 通过编写、与人共同编写计算机方面的书籍以及为获得奖金的作者的书籍作技术编辑, 从而进一步扩大了其在计算机业界的 21 年的事业。他编写的书籍包括有关 PowerPoint、FrontPage、Microsoft SQL Server 和 Oracle SQL 数据库安装、性能调节和编程、Web 服务器安全性及安装以及使用 Active Server Pages、Perl、JavaScript 和 Visual Basic 进行 Web 站点设计与实施等内容。

读者可通过 kki@NVBizNet.com 电子邮件地址与 Konrad 联系。

感 谢

要想感谢在把作者的专业技术和经验转化为读者手中的书稿的过程中所涉及的每一个人，是极其困难的。先请读者花几分钟时间审核一下 Premier Press 和 Argosy 开发组中的专业人士的清单，正是他们使本书成为现实。没有这些人，在读者面前的这些页面中的信息恐怕就不是这个样子了。本书内容的优秀质量是他们努力的直接结果。

首先，让我向 Stacy Hiquet 表达额外的感谢，正是他看到了本书内容的潜力并为保证书稿的出版过程顺利进行提供坚定的援手和耐心。在本书出版的整个过程中，Stacy 总是和颜悦色和性情开朗，这使得写作本书成为一件令人愉快的事情。谢谢你，Stacy，是你给了我进行最好工作的机会。还要感谢 Argosy 的 Daniel Rausch 和 Adriana Lavergne，是他们提供了既具有深厚的技术知识又牢固把握如何将艰深的概念以浅显易懂而又简炼的方式表达出来的编辑人员。如果没有项目经理的极大努力，本书文本和插图的质量和一致性是不可能保证的。

在此还要特别感谢 Lorraine Cooper、Krista Hansing、Elizabeth Agostinelli、David Fields 和编辑团队的其他人员，是他们花费了宝贵的时间来编辑、组织并使技术内容有趣而又易于阅读。在后台工作的编辑人员常常得不到正确的评价。在此要非常感谢这些人，是他们的洞察力、真诚和忘我的贡献，使本书达到了 Premier Press 出版社出版的书籍质量的优秀水准。笔者期待着在以后的项目中再次接受提供满足贵社高标准要求的作品的挑战。

我还要感谢我的朋友 Kris Jamsa，他在似乎无所不包的多种科目上的技术知识令我十分惊讶。Kris，感谢你，是你给了我一些项目上与你一起工作的机会，并将写作标准随着每一本书的出版不断提高。每一个作者都需要优秀的“教练”以写出最好的作品，而每一个人都需要真正的朋友才能度过生活中所遇到的考验与磨难。在改变我的生活（以及写作）使之越来越好上，你既是伟大的教练，也是优秀的朋友。

最后，但绝不是不重要的，我要感谢我的妻子以及我生命中的爱人，Karen King。她的鼓励使我顺利地度过每一章开始时的“空白页”时期，在中间帮助我回答“这些代码为什么不好用？”的问题，是她确保我能以“不错啊”的结果完成每一本书的写作，是她的“只要做下去”的态度使我能够不断地工作下去。Karen，我爱你，而没有你，我也就什么也做不成了！

目 录

精华版序

译者序

作者简介

感谢

第 1 章 理解 SQL 基本知识并创建数据库文件	1
技巧 1 理解数据库的定义	1
技巧 2 理解平面文件	1
技巧 3 理解关系数据库模型	2
技巧 4 理解 Codd 的 12 条关系数据库定义规则	3
技巧 5 理解表	4
技巧 6 理解表名	6
技巧 7 理解列名	6
技巧 8 理解视图	8
技巧 9 理解架构	10
技巧 10 理解域	13
技巧 11 理解约束	13
技巧 12 理解数据定义语言 (DDL)	14
技巧 13 理解数据操纵语言 (DML)	17
技巧 14 理解数据控制语言 (DCL)	18
技巧 15 理解标准 SQL 的日期时间数据类型和 DATETIME 数据类型	19
技巧 16 理解 SQL 的 BIT 数据类型	21
技巧 17 理解 MS-SQL Server 的 IDENTITY 属性	22
技巧 18 理解断言 (Assertions)	23
技巧 19 理解 SQL DBMS 的客户/服务器模型	24
技巧 20 理解 SQL 语句的结构	25
技巧 21 使用 MS-SQL Server Query Analyzer 执行 SQL 语句	28
技巧 22 使用 MS-SQL Server ISQL 在命令行上执行 SQL 语句或是执行存储在 ASCII 文件中的语句	30
技巧 23 在 ISQL 内使用 ED 命令编辑 SQL 语句	32
技巧 24 使用 CREATE DATABASE 语句创建 MS-SQL Server 数据库和事务处理日志	34
技巧 25 使用 MS-SQL Server Enterprise Manager 创建数据库和事务处理日志	36
技巧 26 使用 DROP DATABASE 删除 MS-SQL Server 数据库和事务处理日志	38
技巧 27 理解如何确定 MS-SQL Server 数据库及其事务处理日志的容量	38
技巧 28 理解 MS-SQL Server 的 TempDB 数据库	41

第 2 章 使用 SQL 数据定义语言 (DDL) 创建数据表和其他数据库对象	43
技巧 29 使用 CREATE TABLE 语句创建表	43
技巧 30 使用 MS-SQL Server Enterprise Manager 创建表	44
技巧 31 创建 MS-SQL Server 的临时表	47
技巧 32 使用 Transact-SQL 的 CREATE DEFAULT 语句设置列的默认值	47
技巧 33 使用 MS-SQL Server 的存储过程 sp_bindefault 将用户创建的默认值绑定到表列上	49
技巧 34 在 CREATE TABLE 语句中使用 DEFAULT 子句设置默认列值	50
技巧 35 使用 MS-SQL Server Enterprise Manager 为用户定义的数据类型或表列创建默认值	51
技巧 36 使用 MS-SQL Server Enterprise Manager 创建用户定义的数据类型	52
技巧 37 使用 MS-SQL Server Enterprise Manager 将默认值绑定到数据类型或表列	54
技巧 38 使用 Transact-SQL 的 DROP DEFAULT 语句从数据库中删除默认值	57
技巧 39 使用 ALTER TABLE 语句向表中添加列	60
技巧 40 使用 MS-SQL Server 的 ALTER TABLE、DROP COLUMN 子句删除表列	60
技巧 41 使用 ALTER TABLE 语句改变列的宽度或数据类型	62
技巧 42 使用 ALTER TABLE 语句改变主键和外键	63
技巧 43 使用 CREATE TABLE 语句指定主键	65
技巧 44 使用 CREATE TABLE 语句指定外键约束	66
技巧 45 使用 MS-SQL Server Enterprise Manager Create View Wizard 创建视图	68
技巧 46 理解 DROP VIEW 语句中的 CASCADE 和 RESTRICT 子句	72
第 3 章 使用 SQL 的数据操纵语言 (DML) 在 SQL 表内插入并操作数据	73
技巧 47 使用 INSERT 语句向表中添加行	73
技巧 48 使用 INSERT 语句通过视图插入行	74
技巧 49 使用 MS-SQL Server Enterprise Manager 定义或改变主键约束	76
技巧 50 使用 INSERT 语句向行的特定列中添加数据	77
技巧 51 使用 INSERT 语句将一个表中的行插入另一表	79
技巧 52 将 MS-SQL Server 的 SELECT INTO/BULKCOPY 数据库选项设置为 TRUE 以便加速从表到表的数据转移	82
技巧 53 使用 UPDATE 语句改变列值	83
技巧 54 使用带条件子句的 UPDATE 语句同时改变多行中的值	84
技巧 55 在 UPDATE 语句中使用子查询同时改变多行中的值	86
技巧 56 使用 UPDATE 语句根据另一表中的值改变表的值	86
技巧 57 使用 UPDATE 语句通过视图改变表数据	87
技巧 58 使用 DELETE 语句从表中删除行	89
技巧 59 使用 TRUNCATE 语句从 MS-SQL Server 表中删除所有行	90
技巧 60 使用 DELETE 语句通过视图删除表行	90
第 4 章 处理查询、表达式和总计函数	92
技巧 61 理解 SELECT 语句的结构	92
技巧 62 理解处理 SQL 的 SELECT 语句所涉及的步骤	93
技巧 63 使用 SELECT 语句从一个或多个表的行中显示列	97

技巧 64	使用 SELECT 语句显示列及计算值	98
技巧 65	使用带 WHERE 子句的 SELECT 语句根据列值选择行	100
技巧 66	在 WHERE 子句中使用布尔运算符 OR、AND 和 NOT	100
技巧 67	使用 ORDER BY 子句指定由 SELECT 语句返回行的顺序	101
技巧 68	在 WHERE 子句中使用复合条件 (AND、OR 和 NOT) 根据多个列值 (或计算值) 选择行	104
技巧 69	理解使用比较判式选择行时的 NULL 值	105
技巧 70	使用行值表达式根据多个列值选择表中的行	106
技巧 71	理解子查询	107
技巧 72	使用行值子查询根据多个列值选择表中的行	108
技巧 73	理解表达式	109
技巧 74	理解 SQL 的判式	110
技巧 75	理解集合 (或列) 函数	111
技巧 76	理解 CASE 表达式	112
技巧 77	使用 CASE 表达式更新列值	115
技巧 78	使用 CASE 表达式避免错误条件	116
技巧 79	理解 NULLIF 表达式	117
技巧 80	使用 COALESCE 表达式代替 NULL 值	117
技巧 81	使用 COUNT(*) 总计函数对表中的行数计数	118
技巧 82	使用 COUNT(*) 总计函数对列中的数据值数计数	119
技巧 83	使用 COUNT(*) 总计函数对列中的惟一和重复值计数	120
技巧 84	使用 MS-SQL Server 的 CUBE 和 ROLLUP 运算符总计表的数据	122
技巧 85	使用 MAX() 总计函数找出列中的最大值	124
技巧 86	使用 SUM() 总计函数计算列值的总和	124
技巧 87	使用 AVG() 总计函数计算列中值的平均值	125
技巧 88	使用带 AVG() 函数的 WHERE 子句确定表中所选行的平均值	126
技巧 89	理解 SELECT 语句中的总计函数如何产生单表结果	127
技巧 90	使用 AND 逻辑连接符对表行进行多条件选择	127
技巧 91	使用 OR 逻辑连接符对表行进行多条件选择	128
第 5 章	理解 SQL 的事务处理和事务处理日志	130
技巧 92	理解 SQL 的事务处理过程	130
技巧 93	理解 ANSI/ISO 的事务处理模型	131
技巧 94	理解何时使用 COMMIT 语句	132
技巧 95	使用 ROLLBACK 语句取消对数据库对象所做的改变	134
技巧 96	理解 MS-SQL Server 的事务处理模型	136
技巧 97	在 MS-SQL Server 上使用命名的和嵌套的事务处理	137
第 6 章	使用数据控制语言 (DCL) 建立数据库安全性	141
技巧 98	理解 MS-SQL Server 标准和 Windows NT 的综合安全性	141
技巧 99	使用 MS-SQL Server Enterprise Manager 添加登录和用户	142

技巧 100	使用 MS-SQL Server Enterprise Manager 删除登录和用户	145
技巧 101	理解 MS-SQL Server 的安全角色和组用户安全性	146
技巧 102	理解 MS-SQL Server 的权限	148
技巧 103	理解 SQL 的安全对象和权限	149
技巧 104	使用 MS-SQL Server Enterprise Manager 创建数据库角色	150
技巧 105	使用 MS-SQL Server Enterprise Manager 指定数据库角色权限	151
技巧 106	使用 GRANT 语句的 WITH GRANT OPTION 允许用户向其他用户授予 对数据库对象的访问权	153
技巧 107	理解 REVOKE 语句	154
技巧 108	使用带 CASCADE 选项的 REVOKE 语句删除权限	154
技巧 109	使用 REVOKE 语句的 GRANT OPTION FOR 子句删除 GRANT 权限	156
技巧 110	使用 GRANT SELECT (以及 REVOKE SELECT) 语句控制对数据库对象的访问	157
技巧 111	理解 MS-SQL Server 对 SELECT 权限的列清单扩展	158
技巧 112	使用 GRANT INSERT (以及 REVOKE INSERT) 语句控制对数据库对象的访问	159
技巧 113	使用 GRANT UPDATE (以及 REVOKE UPDATE) 语句控制对数据库对象的访问	160
技巧 114	使用 GRANT REFERENCES (以及 REVOKE REFERENCES) 语句控制 对数据库对象的访问	162
技巧 115	使用 GRANT DELETE (以及 REVOKE DELETE) 语句控制对数据库对象的访问	164
技巧 116	使用 GRANT ALL (以及 REVOKE ALL) 语句授予 (GRANT) 或撤销 (REVOKE) 对数据库对象的权限	166
技巧 117	使用视图将 INSERT 权限限制为只对表中的特定列	168
技巧 118	使用视图将 SELECT 权限限制为只对表中的特定列	169
技巧 119	使用视图扩展 SQL 安全性权限	169
第 7 章	创建索引加快数据引用	171
技巧 120	理解 MS-SQL Server 如何选择用于查询的索引	171
技巧 121	使用 CREATE INDEX 语句创建索引	172
技巧 122	理解 MS-SQL Server 的 CREATE INDEX 语句选项	173
技巧 123	使用 MS-SQL Server Enterprise Manager 创建索引	176
技巧 124	使用 DROP INDEX 语句删除索引	178
技巧 125	理解 MS-SQL Server 的集群索引	178
技巧 126	使用 MS-SQL Server Index Tuning Wizard (索引调节向导) 优化数据库索引	179
第 8 章	使用键字和约束保持数据库的一致性	185
技巧 127	理解单列和复合键字	185
技巧 128	使用 CREATE DOMAIN 语句创建域	186
技巧 129	使用 PRIMARY KEY 列约束惟一地确定表行	187
技巧 130	理解引用完整性检查和外键	188
技巧 131	理解引用数据完整性检查为什么会危害安全性	190
技巧 132	理解引用完整性检查如何限制删除行和表的能力	191
技巧 133	理解引用完整性检查的 INSERT 死锁及解决办法	192

技巧 134	理解 NULL 值与惟一性的相互作用	194
技巧 135	理解如何应用 RESTRICT 规则更新和删除以帮助保持引用完整性	195
技巧 136	理解如何应用 CASCADE 规则更新和删除以帮助保持引用完整性	196
技巧 137	理解如何应用 SET NULL 规则更新和删除以帮助保持引用完整性	197
技巧 138	理解如何应用 SET DEFAULT 规则更新和删除以帮助保持引用完整性	197
技巧 139	使用 Enterprise Manager 在已有表间添加 FOREIGN KEY 关系	198
技巧 140	使用 MATCH FULL 子句保持引用完整性	201
技巧 141	理解 MATCH FULL、MATCH PARTIAL 和 MATCH SIMPLE 子句	202
技巧 142	理解 SET NULL 规则与 MATCH 子句的相互作用	203
技巧 143	使用 NOT NULL 列约束防止列中的 NULL 值	205
技巧 144	使用 UNIQUE 列约束防止列中的重复值	206
技巧 145	使用 CHECK 约束确认列值	206
技巧 146	使用 MS-SQL Server Enterprise Manager 将规则与数据类型或列绑定在一起	208
技巧 147	使用 Transact-SQL 的 CREATE RULE 语句创建 MS-SQL Server 规则	210
技巧 148	使用 MS-SQL Server Enterprise Manager 的 Rule Properties 屏幕改变规则	211
技巧 149	使用 Transact-SQL 的 DROP RULE 语句永久地从数据库中删除规则	212
技巧 150	使用 MS-SQL Server Enterprise Manager 列出并编辑视图	213
技巧 151	使用 CREATE ASSERTION 语句创建多表约束	214
第 9 章	执行多表查询并创建 SQL 视图	216
技巧 152	使用带 FROM 子句的 SELECT 语句进行多表查询	216
技巧 153	使用视图显示一个或多个表或视图中的列	217
技巧 154	使用视图显示一个或多个表的特定行中的列	218
技巧 155	使用 UPDATE 语句通过视图改变多个表中的数据	219
技巧 156	在 CREATE VIEW 语句中使用 CHECK OPTION 子句将视图约束应用于 INSERT 和 UPDATE 语句	220
技巧 157	在 CREATE VIEW 语句中使用 GROUP BY 子句创建显示总结数据的视图	221
技巧 158	使用 CREATE VIEW 语句显示组合两个或多个表的结果	222
技巧 159	使用 UNION 运算符选择出现在任一或全部的两个或多个表中的所有行	223
技巧 160	使用 UNION ALL 运算符选择出现在任一或全部的两个或多个表中的所有行 (包括重复的行)	225
技巧 161	使用 UNION CORRESPONDING 运算符组合来自两个或多个与 UNION 不兼容 的表中的行	226
技巧 162	使用 UNION 运算符组合两条查询的结果	227
技巧 163	使用 ORDER BY 子句对 UNION 运算的结果排序	228
技巧 164	使用 UNION 运算符组合 3 个或 3 个以上的表	228
技巧 165	理解 MS-SQL Server 的事务处理日志放于何处才能改善性能	230
技巧 166	理解多列的 UNIQUE 约束	230
第 10 章	使用函数、参数和数据类型	232
技巧 167	理解实际值	232

技巧 168	使用 SUBSTRING 函数提取部分字符串.....	232
技巧 169	使用 DISTINCT 子句消除行集中的重复.....	233
技巧 170	使用 Transact-SQL 的 STUFF 函数将字符串插入另一字符串.....	235
技巧 171	使用 Transact-SQL 的串接运算符 “+” 在另一字符串尾部添加字符串.....	235
技巧 172	使用 INTERSECT 运算符选择出现在所有两个或多个源表中的行.....	236
技巧 173	使用 EXCEPT 运算符选择出现在一个表而不出现在另一表中的行.....	237
技巧 174	使用 POSITION 函数返回字母或子字符串在字符串中的位置.....	238
技巧 175	使用 CHAR_LENGTH 函数返回字符串变量的长度.....	239
技巧 176	使用 OCTET_LENGTH 函数决定用于保存字符串变量或实际值所需的字节数.....	240
技巧 177	使用 BIT_LENGTH 函数决定用于保存字符串变量或实际值所需的位数.....	241
技巧 178	使用 EXTRACT 函数从 DATETIME 值中提取单个域.....	241
技巧 179	使用 CURRENT_TIME 函数读取当前系统时间.....	242
技巧 180	使用 CURRENT_DATE 函数读取当前系统日期.....	242
技巧 181	使用 CURRENT_TIMESTAMP 函数读取当前系统日期和时间.....	243
技巧 182	理解 MS-SQL Server 的日期和时间函数.....	243
技巧 183	使用 CAST 函数将值从一种数据类型转化为另一种.....	245
技巧 184	使用 CASE 表达式根据列的值选择实际值.....	247
技巧 185	在搜索的 CASE 表达式中使用子查询.....	248
技巧 186	使用 NULLIF 函数将列值设置为 NULL.....	249
技巧 187	使用 CAST 函数比较不同类型列中的值.....	250
技巧 188	使用 CAST 函数从 SQL 向宿主语言中传递值.....	250
技巧 189	理解在 Select 语句中如何使用修饰子句.....	251
第 11 章	使用比较判式和组合查询	253
技巧 190	在 WHERE 子句中使用 BETWEEN 关键词选择行.....	253
技巧 191	在 WHERE 子句中使用 IN 或 NOT IN 判式选择行.....	254
技巧 192	在 LIKE 判式中使用通配符.....	254
技巧 193	在 LIKE 判式中使用转义字符.....	255
技巧 194	使用 LIKE 和 NOT LIKE 比较两个字符串.....	256
技巧 195	理解 MS-SQL Server 对 LIKE 判式中的通配符的扩展.....	256
技巧 196	使用 NULL 判式找出所选列中有 NULL 值的所有行.....	257
技巧 197	理解 UNIQUE 判式.....	258
技巧 198	使用 OVERLAPS 判式决定一个 DATETIME 是否与另一个重叠.....	259
技巧 199	理解 GROUP BY 子句和组合查询.....	259
技巧 200	使用 GROUP BY 子句根据单一列值组合行.....	260
技巧 201	使用 GROUP BY 子句根据多列组合行.....	261
技巧 202	使用 ORDER BY 子句改变由 GROUP BY 子句返回的组中的行序.....	262
技巧 203	使用 MS-SQL Transact-SQL 的 COMPUTE 子句在同一结果表中显示明细及汇总行.....	263
技巧 204	使用 MS-SQL Transact-SQL 的 COMPUTE 和 COMPUTE BY 子句显示多级分类汇总.....	264
技巧 205	理解 GROUP BY 子句如何看待 NULL 值.....	266

技巧 206	使用 HAVING 子句筛选包括在组合查询结果表中的行	267
技巧 207	理解在组合查询中使用 HAVING 子句的 SQL 规则	268
技巧 208	理解 SQL 如何处理 HAVING 子句的 NULL 结果值	269
第 12 章	使用 SQL 的联合语句和其他多表查询	271
技巧 209	使用来自多个 MS-SQL Server 数据库中的表	271
技巧 210	使用 FROM 子句执行多表查询	272
技巧 211	使用 WHERE 子句联合与单列 PRIMARY KEY/FOREIGN KEY 对相关的两个表	272
技巧 212	使用 WHERE 子句联合与复合的 PRIMARY KEY/FOREIGN KEY 对相关的两个表	274
技巧 213	使用 WHERE 子句根据父/子关系联合 3 个或多个表	275
技巧 214	使用 WHERE 子句根据非键字段联合表	276
技巧 215	理解非等价联合	277
技巧 216	在有一个或多个相同列名的联合的多个表中多表查询中使用合格的列名	278
技巧 217	使用带 INTERSECT 运算符的 ALL 关键词在查询结果表中包括重复的行	279
技巧 218	在对非 UNION 兼容表的 INTERSECT 查询中使用 CORRESPONDING 关键词	281
技巧 219	使用没有 WHERE 子句的多表联合生成笛卡尔积	282
技巧 220	使用别名(关联名)作为表名的简写	283
技巧 221	理解 NATURAL JOIN	284
技巧 222	理解条件联合	285
技巧 223	使用 CROSS JOIN 创建笛卡尔积	286
技巧 224	理解列名联合	287
技巧 225	使用 INNER JOIN 选择一个表中与另一表中的行相关的所有行	288
技巧 226	理解 USING 子句在 INNER JOIN 中的作用	289
技巧 227	理解 OUTER JOIN	290
技巧 228	理解 LEFT (RIGHT) OUTER JOIN	291
技巧 229	理解 FULL OUTER JOIN	293
技巧 230	理解 MS-SQL Server 的 OUTER JOIN 记号	294
技巧 231	在单一查询中联合两个以上的表	294
技巧 232	理解非相等的 INNER 和 OUTER JOIN 语句	296
技巧 233	理解 UNION JOIN	296
技巧 234	使用 COALESCE 表达式改善 UNION JOIN 的结果	298
技巧 235	理解 FROM 子句在 JOIN 语句中的作用	299
技巧 236	在多表 JOIN 中使用“*”运算符指明所有或只是某些表中的所有列	300
技巧 237	在单表 JOIN (即自我 JOIN) 中使用表别名	301
技巧 238	理解表的别名	302
技巧 239	理解 ANY 的模糊本质以及 SQL 如何使其表示 SOME	304
技巧 240	使用 EXISTS 而不使用 COUNT(*) 检查子查询是否至少返回一行	305
技巧 241	理解何时使用 ON 子句以及何时使用 WHERE 子句	306
技巧 242	理解如何使用嵌套的查询同时处理多个表	308

第 13 章 理解 SQL 子查询	310
技巧 243 理解在子查询中引用时的主查询列的值	310
技巧 244 在子查询中使用 EXISTS 判式来决定行中是否有满足搜索标准的列值	310
技巧 245 使用关键词 IN 引入子查询	311
技巧 246 使用 ALL 引入返回多个值的子查询	312
技巧 247 在子查询中使用总计函数返回单值	313
技巧 248 理解 WHERE 子句中子查询的作用	313
技巧 249 使用嵌套查询返回 TRUE 或 FALSE 值	314
技巧 250 理解 HAVING 子句中子查询的作用	315
技巧 251 理解 JOIN 语句中关联的和非关联的子查询的执行顺序	317
技巧 252 使用关键词 IN 引入关联子查询来确定有特定值的表列的存在性	318
技巧 253 理解用比较运算符引入的关联子查询	318
技巧 254 将关联子查询用作 HAVING 子句中的过滤器	319
技巧 255 使用关联子查询为 UPDATE 语句选择行	321
第 14 章 理解事务处理隔离级别和并发处理	322
技巧 256 使用带关联子查询的 INSERT 语句创建快照表	322
技巧 257 GRANT 语句授予某人以 DELETE 权限	323
技巧 258 理解 (CASCADE 和非 CASCADE) 取消 GRANT 权限的效果	324
技巧 259 理解如何一起使用 GRANT 和 REVOKE 语句以便在授予权限时节省时间	325
技巧 260 理解并发事务处理问题和隔离级别	325
技巧 261 理解 READ UNCOMMITTED 和作废读取	327
技巧 262 理解 READ COMMITTED 和不可重复读取	327
技巧 263 理解 REPEATABLE READ 和幻影插入	328
技巧 264 理解 MS-SQL Server 的锁定扩大	329
技巧 265 理解死锁以及 DBMS 如何解决死锁	330
技巧 266 理解 SERIALIZABLE 隔离级别	330
技巧 267 理解 REPEATABLE READ 隔离级别	331
技巧 268 理解 READ COMMITTED 隔离级别	332
技巧 269 理解 READ UNCOMMITTED 隔离级别	333
技巧 270 使用 MS-SQL Server Enterprise Manager 显示阻塞和被阻塞的会话	334
技巧 271 使用 MS-SQL Server Enterprise Manager “杀死”对数据库对象保持锁定的进程	335
技巧 272 理解 MS-SQL Server 与 Oracle 上的锁定和事务处理隔离	336
技巧 273 使用 SET TRANSACTION 语句设置事务处理的隔离级别	337
技巧 274 使用 COMMIT 语句使数据库更新成为永久的	338
技巧 275 使用 SET CONSTRAINTS 语句在提交事务处理之前延缓 DEFERRABLE 约束	340
第 15 章 编写外部应用程序来查询与操作数据库数据	341
技巧 276 为开放数据库互连 (ODBC) 连接创建数据源名称 (DSN)	341
技巧 277 向 Visual Basic (VB) 窗体中添加数据控件组件以便提取 SQL 表数据	344
技巧 278 向 Visual Basic (VB) 窗体中添加 MSFlexGrid 控件以显示 SQL 表数据	345

技巧 279	向 Visual Basic (VB) 窗体中添加 Text 和 Button 控件创建向 SQL Server 发送查询的应用程序.....	347
技巧 280	创建用于与 SQL Server 通讯的简单 C++ 外壳程序.....	349
技巧 281	使用 SQLAllocEnv 和 SQLFreeEnv 分配及释放 ODBC 环境资源.....	350
技巧 282	使用 SQLAllocConnect 和 SQLFreeConnect 分配和释放连接句柄和内存资源.....	351
技巧 283	使用 SQLSetConnectOption 为与 SQL Server 的 ODBC 连接设置会话选项.....	353
技巧 284	使用 SQLConnect 和 SQLDisconnect 建立和结束 DBMS 会话.....	354
技巧 285	使用 SQLAllocEnv 和 SQLFreeEnv 分配及释放 SQL 语句句柄和内存资源.....	356
技巧 286	使用 SQLExecDirect 向 DBMS 发送用于执行的 SQL 语句.....	357
技巧 287	使用 SQLFetch 函数从 SQL 数据库中提取数据行.....	359
技巧 288	使用 SQLExtendedFetch 函数创建可更新的游标 (Cursor).....	361
技巧 289	理解行式和列式绑定之间的差别.....	362
技巧 290	使用 SQLSetConnectOption 函数选择在执行 SQL 语句时使用的数据库.....	364
技巧 291	使用 SQLSetPos 函数设置行集中的游标位置.....	365
技巧 292	使用 SQLSetPos 函数的 SQL_UPDATE 选项执行定位更新.....	365
技巧 293	使用 SQLSetPos 函数的 SQL_DELETE 选项执行定位删除.....	366
技巧 294	当 ODBC 驱动程序不支持定位删除时使用 SQLExecDirect 函数删除数据库中的行.....	367
技巧 295	在 ODBC 驱动程序不支持定位更新时使用 SQLExecDirect 函数更新数据库中的列值.....	368
技巧 296	使用 SQLError 函数提取并显示 ODBC 错误代码和错误消息.....	369
技巧 297	在宿主程序变量中处理 NULL 值.....	370
技巧 298	向 Visual Basic (VB) 中添加 DB 函数库 (DBLIB) 功能.....	371
技巧 299	使用 SqlInit() 函数初始化 DB 函数库以及使用 SqlWinExit 例程释放出 SqlInit() 分配的内存.....	373
技巧 300	使用 SqlOpenConnection() 函数登录 MS-SQL Server.....	374
技巧 301	使用 SqlClose() 例程关闭单个 MS-SQL Server 连接或者调用 SqlExit 关闭所有打开的连接.....	375
技巧 302	使用 SqlSendCmd 函数向 MS-SQL Server 发送用于执行的 SQL 语句.....	375
技巧 303	使用 SqlNumCols() 函数确定由查询生成的结果集中的列数.....	376
技巧 304	使用 SqlColName() 函数提取由查询生成的结果集中的列名.....	377
技巧 305	使用 SqlData() 函数从游标中将查询结果提取到应用程序中.....	378
技巧 306	使用 SqlNextRow() 函数在游标行中向前移动.....	379
技巧 307	使用 SqlCmd() 函数建立 SQL 语句批处理.....	380
技巧 308	使用 SqlExec() 函数将 SQL 语句批处理提交给 MS-SQL Server 执行.....	381
技巧 309	使用 SqlResults() 函数提取 SqlExec() 发送的查询结果集.....	381
技巧 310	使用 SqlSend() 提交语句批处理而不必等待 DBMS 完成所有语句的执行.....	382
技巧 311	使用 SqlDataReady() 函数确定 MS-SQL Server 是否完成了 SQL 语句批处理.....	383
技巧 312	使用 SqlCancel() 终止发送到 MS-SQL Server 的语句批处理并清除批结果缓冲区.....	384
技巧 313	使用 SqlCanQuery() 函数在当前结果集中删除剩余 (未被处理的) 行.....	385
技巧 314	使用 SqlUse() 函数为 MS-SQL Server 连接设置当前数据库.....	386
技巧 315	使用 Vbsql1_Error() 例程显示 DBLIB 生成的错误消息.....	387
技巧 316	使用 Vbsql1_Message() 例程显示 MS-SQL Server 生成的错误消息.....	388

技巧 317	使用 SqlColType()函数确定列的数据类型	389
技巧 318	使用 SqlDatLen()函数确定储存在 DBLIB 缓冲区列中的数据的字节数	390
技巧 319	在 Visual Basic 应用程序中给宿主变量指定 NULL 值	391
技巧 320	使用 SqlSetOpt()设置行缓冲区的大小以使用 SqlGetRow()随机提取行	392
技巧 321	使用 SqlGetRow()函数在 DBLIB 查询结果缓冲区中选择当前行	393
技巧 322	使用 SqlClrBuff()函数在 DBLIB 查询结果缓冲区中为附加行腾出空间	394
技巧 323	理解 MS-SQL Server 的 SELECT 语句中的 FOR BROWSE 子句	395
技巧 324	理解 DBLIB 为什么不支持定位 UPDATE 和 DELETE 语句	395
技巧 325	理解 DBLIB 浏览模式的函数	396
技巧 326	使用 SqlQual()函数为 DBLIB 浏览模式的 UPDATE 或 DELETE 语句生成 WHERE 子句	397
技巧 327	执行 DBLIB 浏览模式的 DELETE 语句	397
技巧 328	执行 DBLIB 浏览模式的 UPDATE 语句	398
技巧 329	用 DBLIB API 执行动态 SQL 查询	399
第 16 章	通过游标提取和维护数据	402
技巧 330	理解游标的目的	402
技巧 331	使用 DECLARE CURSOR 语句定义游标	402
技巧 332	使用 OPEN 语句创建游标	403
技巧 333	使用 ORDER BY 子句改变游标中行的顺序	404
技巧 334	在游标中包含计算好的值作为列	405
技巧 335	使用 FOR UPDATE 子句指定游标可修改底层表的哪些列	405
技巧 336	使用 FETCH 语句从游标中的行提取列值	406
技巧 337	把游标的当前行指针预先定向到从当前行获取列值	407
技巧 338	理解基于游标的定位 DELETE 语句	408
技巧 339	理解基于游标的定位 UPDATE 语句	409
技巧 340	使用索引改变游标中行的顺序	410
技巧 341	使用 @@FETCH_STATUS 利用 WHILE 循环处理游标中的行	410
技巧 342	使用 DEALLOCATE 语句删除游标并释放其服务器资源	411
技巧 343	理解 DECLARE CURSOR 语句的 Transact-SQL 扩展句法	412
技巧 344	使用 @@CURSOR_ROWS 系统变量确定游标中的行数	413
第 17 章	理解触发器	415
技巧 345	理解何时用 CHECK 约束代替触发器	415
技巧 346	理解嵌套游标	415
技巧 347	理解当前日期和时间的值是在语句开始执行时设置的	417
技巧 348	用 CREATE TRIGGER 语句创建触发器	418
技巧 349	理解 INSERT 触发器	420
技巧 350	理解 DELETE 触发器	421
技巧 351	理解 UPDATE 触发器	422
技巧 352	用 UPDATE 触发器改变 PRIMARY KEY/FOREIGN KEY 对的值	423
技巧 353	用触发器发送 E-mail 消息	425

技巧 354	用 MS-SQL Server Enterprise manager 显示或修改触发器	425
技巧 355	用 ALTER VIEW 语句修改视图	427
技巧 356	用 ALTER TABLE 语句改变列的数据类型	428
第 18 章	处理 Blobs 数据和文本	429
技巧 357	理解由二进制和字符大对象 (BLOB) 的处理带来的挑战	429
技巧 358	理解 MS-SQL Server 的 BLOB (TEXT、NTEXT 和 IMAGE) 数据处理过程	430
技巧 359	用 INSERT 或 UPDATE 语句把数据放到 BLOB 数据类型的列中	431
技巧 360	用 Transact-SQL WRITETEXT 语句把数据放到 TEXT、NTEXT 或 IMAGE 列中	432
技巧 361	用 Transact-SQL UPDATETEXT 语句改变 TEXT、NTEXT 或 IMAGE 列的内容	433
技巧 362	用 READTEXT() 函数读取 TEXT、NTEXT 或 IMAGE 列中的部分 (或全部) 数据	434
技巧 363	用 MS-SQL Server 的 TEXTVALID() 函数确定文本指针是否有效	435
技巧 364	用 PATINDEX() 函数返回 BLOB 中第一次出现的地址	435
技巧 365	用 DATALENGTH() 函数返回 BLOB 中的字节数	436
技巧 366	理解 TEXTSIZE 选项和 @@TEXTSIZE() 函数	437
第 19 章	使用 MS-SQL Server 信息架构视图	438
技巧 367	理解信息架构	438
技巧 368	理解信息架构的 CHECK_CONSTRAINTS 视图	438
技巧 369	理解信息架构的 COLUMN_DOMAIN_USAGE 视图	439
技巧 370	理解信息架构的 COLUMN_PRIVILEGES 视图	440
技巧 371	理解信息架构的 COLUMNS 视图	440
技巧 372	理解信息架构的 CONSTRAINT_COLUMN_USAGE 视图	442
技巧 373	理解信息架构的 CONSTRAINT_TABLE_USAGE 视图	443
技巧 374	理解信息架构的 DOMAIN_CONSTRAINTS 视图	444
技巧 375	理解信息架构的 DOMAINS 视图	444
技巧 376	理解信息架构的 KEY_COLUMN_USAGE 视图	445
技巧 377	理解信息架构的 PARAMETERS 视图	446
技巧 378	理解信息架构的 REFERENTIAL_CONSTRAINTS 视图	447
技巧 379	理解信息架构的 ROUTINES 视图	449
技巧 380	理解信息架构的 SCHEMATA 视图	451
技巧 381	理解信息架构的 TABLE_CONSTRAINTS 视图	451
技巧 382	理解信息架构的 TABLE_PRIVILEGES 视图	452
技巧 383	理解信息架构的 TABLES 视图	453
技巧 384	理解信息架构的 VIEW_COLUMN_USAGE 视图	454
技巧 385	理解信息架构的 VIEW_TABLE_USAGE 视图	454
技巧 386	理解信息架构的 ROUTINE_COLUMNS 视图	455
技巧 387	理解信息架构的 VIEWS 视图	456
技巧 388	用 MS-SQL Server Enterprise Manager 查看信息架构视图的内容	457
技巧 389	理解 MS-SQL Server 系统数据库表	459
技巧 390	定义数据库的物理位置	460

技巧 391	向已有数据库添加文件和文件组.....	461
技巧 392	通过创建联接表视图简化多表查询.....	462
技巧 393	理解 CREATE VIEW 语句中的 WITH SCHEMABINDING 子句.....	463
第 20 章	监测及提高 MS-SQL Server 的性能.....	465
技巧 394	理解多处理器 Windows NT 系统上的 MS-SQL Server 多任务与多线程.....	465
技巧 395	用 MS-SQL Server 的 PRIORITY BOOST 配置选项把服务器线程的优先权从 7 增加到 13..	466
技巧 396	理解 NT Server 的性能监视器的图表视图.....	467
技巧 397	理解 NT Server 性能监视器的报告视图.....	468
技巧 398	理解 NT Server 性能监视器的警报视图.....	470
技巧 399	使用 CREATE SCHEMA 语句创建表并授予对此表的访问权限	471
技巧 400	建立 NT Server 性能监视器日志以帮助优化 MS-SQL Server	473
技巧 401	用 NT 性能监视器查看性能日志文件	474
技巧 402	配置 Windows NT 的应用程序事件日志	476
技巧 403	显示 Windows NT 应用程序事件详情并清除应用程序事件日志.....	477
技巧 404	用 MS-SQL Server 服务管理器启动 MS-SQL Server.....	478
技巧 405	理解如何恢复 MS-SQL Server 数据库.....	479
技巧 406	理解 MS-SQL Server 优化器提示.....	484
技巧 407	用 MS-SQL Server 的 SHOWPLAN_TEXT 选项显示语句的执行计划.....	486
技巧 408	理解显示语句执行计划和状态的 MS-SQL Server SHOWPLAN_ALL 选项	487
技巧 409	使用 MS-SQL Server SQL Query Analyzer 的 SHOWPLAN 选项.....	488
技巧 410	用 MS-SQL Server SETUSER 语句测试用户对数据库对象的访问权限.....	490
第 21 章	使用存储过程	491
技巧 411	理解存储过程.....	491
技巧 412	使用 CREATE PROCEDURE 语句创建存储过程	492
技巧 413	用 EXECUTE 语句调用存储过程	494
技巧 414	使用存储过程参数返回值.....	495
技巧 415	用关键词 RETURN 从存储函数中返回一个值.....	497
技巧 416	在存储过程中使用游标.....	499
技巧 417	使用 CREATE FUNCTION 语句创建存储函数.....	502
技巧 418	使用 MS-SQL Server Enterprise Manager 查看或修改存储过程或函数	505
技巧 419	使用 Transact-SQL 关键词 DECLARE 和 SELECT 在存储过程中定义变量并为其赋初始值	507
第 22 章	修理及维护 MS-SQL Server 数据库文件	509
技巧 420	理解 MS-SQL Server 的 Database Consistency Checker (DBCC, 数据库一致检查器)	509
技巧 421	理解 DBCC 的维护语句.....	510
技巧 422	理解 DBCC 的杂项语句.....	514
技巧 423	理解 DBCC 的状态语句.....	515
技巧 424	理解 DBCC 的确认语句.....	520
第 23 章	编写高级查询及子查询.....	526
技巧 425	理解对用作比较运算符判式的子查询的限制.....	526

技巧 426	使用视图允许子查询中的工作表自我联合	527
技巧 427	使用临时表删除重复数据	528
技巧 428	使用临时表从多表中删除行	529
技巧 429	使用 UPDATE 语句根据另一个表中的值设置表中的值	531
技巧 430	优化 EXISTS 判式	532
技巧 431	使用 ALL 判式把两个查询合二为一	534
技巧 432	使用 EXISTS 判式检查表中的重复行	535
技巧 433	把表内容和函数结果合并	536
技巧 434	使用视图显示汇总级别的层次	537
技巧 435	使用带标量子查询的 SELECT 语句显示流水总计	538
技巧 436	使用 EXCEPT 判式确定两表差异	539
技巧 437	使用 EXISTS 判式生成两表的交集	540
第 24 章	探索 MS-SQL Server 的内建存储过程	542
技巧 438	使用 sp_detach_db 和 sp_attach_db 在 MS-SQL Server 上删除和添加数据库	542
技巧 439	使用 MS-SQL Server 的存储过程 sp_addtype 和 sp_droptype 添加和删除用户定义的数据类型	543
技巧 440	使用 sp_help 显示数据库对象属性	544
技巧 441	使用 sp_helptext 显示定义存储过程、用户定义函数、触发器、默认值、规则或者视图的文本	546
技巧 442	用 sp_depends 显示定义视图的表和（或）视图	546
技巧 443	使用 sp_helpconstraint 显示有关表约束的信息	547
技巧 444	使用 sp_pkeys 显示表的 PRIMARY KEY 信息	549
技巧 445	使用 sp_fkeys 显示关于引用表的 PRIMARY KEY 的外键信息	550
技巧 446	使用 sp_procoption 控制 MS-SQL Server 启动时运行的存储过程	552
技巧 447	使用 sp_spaceused 显示分配给数据库或单独的数据库对象的已用与未用空间量	553
技巧 448	使用 sp_helptrigger 显示有关表上的触发器信息	554
技巧 449	使用 sp_who 和 KILL 命令控制运行在 MS-SQL Server 上的进程	556
技巧 450	使用 sp_lock 显示数据库所掌握的锁定信息	557
技巧 451	使用 sp_password 改变账户密码	559
第 25 章	通过 Internet 处理 SQL 数据库中的数据	561
技巧 452	使用 sp_makewebtask 创建生成 Web 页面的任务	561
技巧 453	为 MS-SQL Server 查询结果创建 Web 页面模板	566
技巧 454	格式化由 MS-SQL Server 存储过程创建的 Web 页面上的查询结果表	569
技巧 455	使用 sp_makewebtask 在链接的 Web 页面上显示 IMAGE 和 TEXT 数据	571
技巧 456	使用内建的存储过程启动或删除 Web 任务	574
技巧 457	使用 MS-SQL Server Web Assistant Wizard 创建执行存储过程的 Web 任务	575
技巧 458	下载并安装 PHP	580
技巧 459	建立数据源（DSN）与 SQL DBMS 的连接	581
技巧 460	下载、安装并使用 MyODBC 驱动程序与 MySQL 数据库连接	584

技巧 461	与 MS-SQL Server 或 MySQL DBMS 建立无 DSN 的连接	587
技巧 462	使用 ADO Connection 对象执行 SELECT 语句设置访问 Web 站点的用户名/密码	589
技巧 463	在 Web 页面上的 HTML 表中显示查询结果	591
技巧 464	编写可重用的 PHP 例程在 Web 页面上显示查询结果	593
技巧 465	通过 HTML 表单提交 SQL 查询	598
技巧 466	使用 HTML 表单向 SQL 表中插入数据	602
技巧 467	通过 HTML 表单更新及删除数据库数据	604
技巧 468	从脚本内调用存储过程	608
技巧 469	使用 VBScript 处理 Recordset 对象	609
技巧 470	通过 Internet 处理 SQL 的事务处理过程	613
技巧 471	创建与 MS-SQL Server 的虚拟连接	615
技巧 472	使用 HTTP 执行 SQL 语句	619
技巧 473	使用 XML 架构利用 HTTP 提交查询并使用 XSL 样式表来格式化查询结果	621
技巧 474	显示保存在 SQL 表内的图像数据	626

第 1 章 理解 SQL 基本知识并创建数据库文件

技巧 1 理解数据库的定义

顾名思义，数据库包括数据。数据被组织为描述物理和概念上的对象的记录（records）。相关的数据库记录组合成表（tables）。例如，顾客记录可由数据项或诸如名称、顾客编号、地址、电话号码、信用评级、生日、周年纪念日之类的属性组成。简短地说，顾客记录是能惟一地标识一个人（或其他事务）的一组属性或是特性。顾客表是顾客记录的集合。

平面文件（有关内容将在技巧 2“理解平面文件”中加以讨论）只包括数据，而数据库既包括数据也包括元数据。元数据是描述以下事项的：

- 每条记录中的字段（表中的列）
- 在每个表中的记录的位置、名称和号码
- 用于找出表中记录的索引
- 定义可赋予单条记录属性（或字段）的值范围的值约束
- 定义什么样的记录可添加到表，并限制记录删除方式的键约束；也包括不同数据库表记录之间的关系

数据库中的数据组织为多个表内的有关系的记录，而数据库的元数据放在名为数据字典（data dictionary）的独立的表中。

简短地说，一个数据库被定义为自描述的组织成表的记录的集合。由于数据库在数据字典表内包括描述每个记录（或表中的行）中的字段（或属性）以及将相关的记录组合成表的结构元数据，所以称为自描述的。

技巧 2 理解平面文件

平面文件是数据记录的集合。如下所示，人们看到的是一行接一行的数据：

010000BREAKFAST JUICES	F00.000000
010200TREE TOP APPLE JUICE	12OZF01.100422
010400WELCHES GRAPE JUICE	12OZF00.850198
010600MINUTE MAID LEMONADE	12OZF00.850083
010800MINUTE MAID PINK LEMONADE	12OZF00.890099
011000MINUTE MAID ORANGE JUICE	12OZF01.260704
011400MINUTE MAID FRUIT PUNCH	12OZF00.820142
011600CAMPBELLS CAN TOMATO JUICE	46O2G01.200030
020000FAMOUS BRAND CEREALS	G01.200000
020200GENERAL MILLS CHEERIOS	15O2G03.010050

请看一下平面文件清单，可以发现，文件只包括数据。空格用来将一个字段与另一字段分隔开来，而每一个非空行就是一条记录。每种读取数据文件的应用程序必须“了解”每个“字段”中的字符个数以及数据表示什么。这样一来，程序必须有几行代码，将一行中的前 6 个字

符读作物品号码，而接下来的 32 个字符是描述，其后的 1 个字符是部门指示符，再接下来的 5 个字符是销售价格，最后的 4 个数字是每周所发送物品的平均计数。

平面文件和数据库之间的主要差别就在于，平面文件由操作系统的文件管理系统管理，而且不包括对其内容的描述。因此使用平面文件的应用程序必须包括平面文件记录布局的定义，指定要执行的活动（读取、写入、删除）的代码以及低级的与操作系统有关的执行程序意图的命令。另一方面，数据库是由 DBMS 管理的，由 DBMS 处理操作数据库文件数据的低级命令以及指定做什么和如何做的命令。处理数据库的程序只指定要完成什么而将如何完成的细节留给了 DBMS。

技巧 3 理解关系数据库模型

在关系数据库中，所有数据都组织到表中，如图 1.1 所示。

顾客表

CUST_NO	LAST_NAME	FIRST_NAME
10	FIELDS	SALLY
11	CLEAVER	WARD

销售人员表

SALESMAN_NO	LAST_NAME	FIRST_NAME
5	KING	KAREN
6	HARDY	ROBERT

产品表

PRODUCT_NO	DESCRIPTION	SALES PRICE	INV COUNT
7	100 WATT SPEAKER	75.00	25
8	DVD PLAYER	90.00	15
9	AMPLIFIER	450.00	305
10	RECEIVER	750.00	25
11	REMOTE CONTROL	25.00	15
12	50 DVD PACK	500.00	25

订单表

ORDER_NO	DEL DATE	PRODUCT_NO	SALESMAN_NO	CUST_NO
101	01/15/2000	7	5	10
102	01/22/2000	12	5	11
103	03/15/2000	7	6	10
104	04/05/2000	12	7	12
105	07/05/2000	9	6	11
106	08/09/2000	7	8	11

图 1.1 具有与其他 3 个表的关系的 ORDER_TABLE 表

关系数据库模型使用共用的列来建立表间的关系。如图 1.1 所示，CUSTOMER、SALESMAN 和 PRODUCT 表是没有关系的，因为其中没有共同的列。但是，ORDER 表与 CUSTOMER 表通过 CUSTOMER_NO 列发生了关系。由此，ORDER 表行与 CUSTOMER 表行有了关系，因为 CUSTOMER_NO 列在两个表中有相同的值。图 1.1 表明 CUSTOMER 10 拥有 ORDER 101 和 ORDER 103，因为这两个 ORDER 行的 CUST_NO 列等于 CUSTOMER 表中的 CUSTOMER_NO 列。SALESMAN 和 PRODUCT 表都以同样的方式与 ORDER 表有关。SALESMAN 表与 ORDER 表有关系是通过共同的 SALESMAN_NO 表示出来的，而 PRODUCT 表与 ORDER 表有关系是通过 PRODUCT_NO 列表示出来的。

当处理关系数据库时，可添加表并向表内添加列，以创建与新表的新关系，而不必重新编译已有的应用程序。惟一需要重新编译应用程序的时候是在删除或改变了由程序使用的列时。因而，如果想要使 SALESMAN 表中的条目与一个顾客（在 CUSTOMER 表中）有关系，只需要向 CUSTOMER 表中添加一个 SALESMAN_NO 列即可。

技巧 4 理解 Codd 的 12 条关系数据库定义规则

1970 年，Edgar F. Codd 博士在 Communications of the ACM 上的一篇名为 A Relational Model of Data for Large Shared Data Banks（大型共享数据库数据的关系模型）的文章中发表了第一个关系数据库的理论模型。它提出了定义关系数据库的 Codd 的 12 条规则：

1. 信息规则。关系数据库中的所有信息必须以一种且只能以一种方式来表示，即通过表行中的列值来表示。SQL 满足这条规则。

2. 保证访问规则。每个数据（或行中的单独列值）必须是可逻辑寻址的，可通过指定表名、主键值和列名来寻址。当对一个数据项寻址时，表名确定哪个数据库表包括该数据项，列确定指定表行中的数据项，而主键确定表中的单个行。SQL 对于带有主键的表遵循这一规则，但并不要求表具有键。

3. Null 值的系统处理。关系数据库管理系统必须能够以独立于数据类型的系统方式来表示缺少或是不可用的信息，这与用于表明空白字符串或是空字符串是不同的，与零或任何其他数字也是截然不同的。SQL 使用 NULL 既可表示缺少也可表示不可用的信息，正如以后所学到的，NULL 并不是零也不是空白的字符串。

4. 基于关系模型的动态在线目录。数据库目录（或是描述）是以与普通数据（使用表）一样的方式来表示的，因此授权用户可使用同样的关系语言来处理在线目录和正规的数据。SQL 是通过其列描述数据库结果的系统表来实现的。

5. 广泛的数据子语言规则。系统可支持不止一种语言，但至少有一种语言必须具有基于字符串的精确定义的句法，而且既可用于交互式也可用于应用程序内。该种语言必须支持：

- 数据定义
- 视图定义
- 数据操纵（更新和提取）
- 安全性
- 完整性约束
- 事务管理操作（Begin、Commit、Rollback）

SQL 数据操纵语言（DML，Data Manipulation Language）（既可交互使用，也可用在程序中）具有执行所有需要操作的语句。

6. 视图更新规则。所有理论上可更新的视图必须可由系统来更新（视图是给用户以不同“图景”或表示数据库结构的虚拟表）。SQL 并不完全满足这条规则，因为它将可更新的视图限制为那些基于对单个表的查询，而没有 GROUP BY 或 HAVING 子句。它还没有合计函数，没有计算的列也没有 SELECT DISTINCT 语句。另外，视图必须包括表的一个键，而且从视图中排除的任何列必须是在基表中的可成为 NULL 的列。

7. 高级插入、更新和删除。系统必须支持多行和表（一次一个数据集）的插入、更新和删除操作。SQL 通过在插入、更新和删除时将行视作数据集而满足这一规则的。

8. 物理数据独立性。应用程序和交互式数据库访问方法不必由于物理存储设备或是从设备中用于提取数据的方法的改变而改变。SQL 很好地满足这一规则。

9. 逻辑数据独立性。如果表的变化是以保留原始表值的方式发生的，则应用程序和交互式数据库访问方法不必变化。SQL 满足这一要求——查询和语句所采取动作的结果并不依赖于行中列的排列、表中行的位置或用来表示计算机系统内表的结构。

10. 完整独立性。所有与特定关系数据库有关的完整性约束在关系子语言中必须是可定义的，必须在应用程序之外指定，且存储在数据库类别中的。SQL-92 具有完整独立性。

11. 分布独立性。应用程序和最终用户不应该了解数据库数据是存在于单个位置上，还是位于分布在网络上的许多计算机上的。因此，数据库语言必须能够使用相同的命令来查询和操作在本地和远程计算机系统上分布的数据。分布式 SQL 数据库产品对于是否满足这一准则还没有定论。

12. 非子版本规则。如果系统提供低级（一次一个记录）的接口，则低级语句不能用来绕过在高级（一次一个数据集）语言中表达的完整性规则和约束。SQL-92 遵守这一规则。虽然人们可编写影响单独表行的语句，但系统仍然强制实行安全性和引用完整性规则。

正如没有 SQL DBMS 符合所有 SQL-92 标准中的所有规范一样，也没有一个商业上可用的关系数据库遵循 Codd 的所有 12 条规则。但是 Codd 规则从历史的角度来看是重要的，这些规则确实有助于决定一个 DBMS 是否是基于关系模型的。

技巧 5 理解表

一个 SQL 表由标量（单个值）数据按行和列排列组成。关系数据库表具有下列组成：

- 一个惟一的表名
- 表中的每一列有惟一的名称
- 至少有一列
- 数据类型、域以及指定表中每一列的数据类型和值范围的约束
- 结构，其中表中每一列的数据在每行中都有同样的意义
- 代表物理或逻辑实体的零行或多行

当命名表时，请记住，不能有两个表具有相同的名称。但是，在 SQL 数据库中，表名仅在某一单个用户所创建（或拥有）的表中要求惟一性。由此，如果两个用户——例如 Joe 和 Mark——要在 SQL 数据库中创建表，两者都可创建名为 Stocks 的表。但是，两人不能在同一架构内创建两个名为 Stock 的表。在技巧 6“理解表名”中，读者将学习有关名称以及 DBMS 如何使用拥有者的名字和架构名来使名称在数据库中的表中惟一的知识。现在先讲如何给表起名。在选择表名时，请分析一下打算包括在表中的列，并为表起一个对其内容有总结性的名称。例如，图 1.2 包括了处理电话呼叫数据的列：PHONE_REP_ID（谁打的电话）、PHONE_NUMBER（被叫电话号码）、DATE_TO_CALL 和 TIME_TO_CALL（打电话时的日期和时间）、HANGUP_TIME（电话结束的时间）以及 DISPOSITION（电话打完后发生了什么）。列名指明该表将包括电话呼叫数据。因而 CALL_HISTORY 是合适的表名，因为该名称描述了可从表中提取的数据的类型。

通常每一行的列中的数据都详细说明了该行所代表的实体的属性。请注意，在图 1.2 中显示的 CALL_HISTORY 表中的列都描述了电话呼叫的某种属性。

CALL HISTORY 表

PHONE REP ID	PHONE NUMBER	DATE TO CALL	TIME TO CALL	DATE CALLED	TIME CALLED	HANDOUT FINE	DISPOSITION

图 1.2 电话呼叫历史记录数据的关系数据库表

所有的关系数据库表至少有一列（表可以没有行，但至少必须有一列）。SQL 标准并未指定最大的列数，但大多数商业数据库通常将表中的列数限制为 255。类似地，SQL 标准对表中可包括的行数没有限制。结果大多数 SQL 产品允许表可增长到耗尽可用的磁盘空间为止，或者，如果施加限制的话，可将此数设置为几十亿。

表中列的顺序对数据库的 SQL 查询结果没有影响。但是在创建表时，确实必须指定列的顺序，为每一列起一个惟一的名称，指定每一列将包括的数据类型以及对列值指定任何约束（或限制）。为了创建图 1.2 所示的表，可以使用以下 SQL 语句：

```
CREATE TABLE CALL_HISTORY
(PHONE_REP_ID CHAR(3) NOT NULL,
PHONE_NUMBER INTEGER NOT NULL,
DATE_TO_CALL DATE,
TIME_TO_CALL INTEGER,
DATE__CALLED DATE NOT NULL,
TIME_CALLED INTEGER NOT NULL,
HANGUP_TIME INTEGER NOT NULL,
DISPOSITION CHAR(4) NOT NULL)
```

当向下查看关系数据库表中的列时，人们将注意到，列数据是自我描述的，也就是说列中的数据在列中的每行中有同样的意义。这样一来，虽然列的顺序对查询来说是非实质性的，但表必须有某种对列的排列，而且从一行到另一行不能改变。在创建了该表之后，可使用 ALTER TABLE 命令对列重新排列；这样做对表中数据的查询没有什么影响。

表中的每一列都具有惟一的名称，这个名称是在执行 **CREATE TABLE** 语句时赋予列的。在当前的例子中，列标题名显示在图 1.2 中的每一列的顶部。请注意，DBMS 在表中是从左向右赋予列名的，而且是按照出现在 **CREATE TABLE** 语句中的顺序进行的。表中的所有列必须具有不同（惟一）的名称。但是，同样的列名可以出现在多个表中。比如，在 **CALL_HISTORY** 表中只能有一个 **PHONE_NUMBER** 列，但可在另一表中也有 **PHONE NUMBER** 列，例如在

CALLS_TO_MAKE 表中。

技巧 6 理解表名

在为表选择名称时，一定要使其简短但对表内包括的数据具有描述性。最好使名称短小，并使名称具有描述性以便于记忆。如果正在使用个人或部门数据库，通常用户有权随便对表命名。SQL 并未指定名称以何种字母或字母集开始。仅有的要求是表名必须对拥有者是惟一的。如果大公司对表名有所限制，可以按部门组织表以避免名称冲突。例如，在大型组织机构内，销售表可能都会以 SALES_ 开头，用于人力资源的表可能会以 HR_ 开头，而用于顾客服务的表可能会以 SERVICE_ 开头。再次强调，SQL 只要求对拥有者表名惟一，除此之外对表名没有限制。

为了创建表，必须登录到 SQL DBMS，用户名必须授权才能使用 CREATE TABLE 语句。登录到 DBMS 之后，系统识别用户名并自动使用该用户拥有自己所创建的表。因而，如果在多用户环境下工作，DBMS 确实可以有不止一个表的名称为 CUSTOMER，但对任何单个用户来说，只能有一个 CUSTOMER 表。例如，假设 DBMS 用户 Karen 和 Konrad 每人创建了一个 CUSTOMER 表。DBMS 将自动地将拥有者名（在默认情况下，表的拥有者是创建表的人的用户 ID）添加到表名上，以形成“限定的表名”（qualified table name），然后将其保存在系统目录中。这样一来，Karen 的 CUSTOMER 表会以 KAREN.CUSTOMER 的形式保存在系统目录中，而 Konrad 的表会以 KONRAD.CUSTOMER 的形式加以保存。结果，在系统目录中的所有表名仍然是惟一的，即使 Konrad 和 Karen 都执行相同的 SQL 语句：CREATE TABLE CUSTOMER。

当用户登录到 DBMS 并键入引用表名的 SQL 语句时，DBMS 将认为所要引用的是其自己创建的表。由此，如果 Konrad 登录并键入 SQL 语句 SELECT * FROM CUSTOMER，DBMS 将返回 KONRAD.CUSTOMER 表内的所有行的所有列的值。类似地，如果 Karen 登录并执行同样的语句，DBMS 将显示 KAREN.CUSTOMER 表中的数据。如果另一用户（例如 Mark）登录并键入 SQL 语句 SELECT * FROM CUSTOMER，而没有首先创建 CUSTOMER 表，系统将返回一条错误，因为 DBMS 并不具有名为 MARK.CUSTOMER 的表。

为了使用由另外的用户所创建的表，必须获得适当的授权（访问权），而且必须键入限定的表名。限定的表名指定表拥有者的名称，后面跟一句号（.），然后才是表名（形如 <owner>.<table name>）。在前面的例子中，如果 Mark 具有适当的授权，他可键入 SQL 语句 SELECT * FROM KONRAD.CUSTOMER，以便显示 Konrad 的 CUSTOMER 表中的数据，或者键入 SELECT * FROM KAREN.CUSTOMER 语句，显示 Karen 的 CUSTOMER 表中的内容。凡是在 SQL 语句中可以使用表名的地方都可以使用限定的表名。

SQL-92 标准进一步扩展了 DBMS 的能力，通过允许一个用户在架构内创建表使之可处理重复的表。（在技巧 9“理解架构”中将学习有关架构的更多内容。而在技巧 399“使用 CREATE SCHEMA 语句创建表并授予对此表的访问权限”中将学习有关在架构内创建表的内容。）在一种架构内创建的表的完全限定的表名的形式是架构名后跟句号（.）然后再跟表名（例如 <schema>.<table name>）。这样一来，单独的用户可在不同的架构内创建多个相同名称的表。

技巧 7 理解列名

SQL DBMS 在系统目录中将表名和列名保存在一起。列名在表内必须是惟一的，但可出

现在多个表中。例如，在 STUDENT 表和 CLASS_SCHEDULE 表中都可能有 STUDENT_ID 列。但是，在任何一个表中都不能有多个 STUDENT_ID 列。在选择列名时，请使用简短、惟一（对所创建的表来说）且可概括列中将包括的数据种类的名称。SQL 规范并未限制对列名的选择方式，但是，具有描述性的列名，可使之易于在编写从表中提取数据的 SQL 语句时了解应该使用哪些列。

当在 SQL 语句中指定列名时，如果列名对语句中的单个表来说是惟一的，则 DBMS 可决定要引用的表。例如，假设有两个表都有如图 1.3 所定义的列名。

DBMS 决定在以下 SQL 语句中显示哪一列上没有什么问题：

```
SELECT
    STUDENT_ID,    STUDENT_NAME,    SUBJECT,
TEACHER_NAME
FROM
    STUDENT, TEACHER
WHERE
    CLASS1_TEACHER = TEACHER_ID
```

由于 STUDENT_ID 和 STUDENT_NAME 只出现在 STUDENT 表中，DBMS 将显示 STUDENT 表中的 STUDENT_ID 和 STUDENT_NAME 的值。类似地，SUBJECT 和 TEACHER_NAME 只能在 TEACHER 表中找到，因而 DBMS 在执行 SELECT 语句时将显示 TEACHER 表中的 SUBJECT 和 TEACHER_NAME 列的信息。这样一来，如果在 SQL 语句中从不止一个表中使用一些列，而且没有一个列名出现在 FROM 子句中列出的多个表中的话，DBMS 可以找出哪个列名引用了哪个表。

如果在一条 SQL 语句中使用不止一个表，希望显示来自一个或多个具有相同列名的列中的数据，则对每个重复的列必须使用限定的列名。限定的列名的形式是，表名后跟句号 (.) 再跟列名。由此，如果希望列出学生的电话号码（在 STUDENT 表中的 PHONE_NUMBER 列中），可以使用以下 SQL 语句：

```
SELECT
    STUDENT_ID, STUDENT_NAME, STUDENT.PHONE_NUMBER, SUBJECT,
TEACHER_NAME
FROM
    STUDENT, TEACHER
WHERE
    CLASS1_TEACHER = TEACHER_ID
```

如果只在 STUDENT_NAME 后指定 PHONE_NUMBER，DBMS 就不知道是显示学生的电话号码还是老师的电话号码，因为两个表中都有名为 PHONE_NUMBER 的列。通过使用限定的列名（在此例中是 STUDENT.PHONE_NUMBER），就可不仅指定希望获得数据的列，而且还指定了 DBMS 要使用的表。一般来说，可在 SQL 语句中任何可出现非限定列名的地方使用限定列名。

必须在 SQL 语句中对不属于自己的表使用限定的表名，即在拥有者名后跟句号 (.)，然后跟

学生表	
STUDENT_ID	
STUDENT_NAME	
STREET_ADDRESS	
CITY	
STATE	
ZIP_CODE	
PHONE_NUMBER	
CLASS1_TEACHER	
CLASS2_TEACHER	
CLASS3_TEACHER	
CLASS4_TEACHER	

教师表	
TEACHER_ID	
TEACHER_NAME	
SUBJECT	
PHONE_NUMBER	

图 1.3 STUDENT 表和 TEACHER 表都有重复列名的例子

表名。在当前的例子中，如果用户拥有 TEACHER 表，但未创建 STUDENT 表（或是 Konrad 创建的该表，为用户授权可访问此表，但没有把所有权授予用户），现在可将 SQL 语句修改如下：

```
SELECT
    STUDENT_ID, STUDENT_NAME, KONRAD.STUDENT.PHONE_NUMBER,
    SUBJECT, TEACHER_NAME
FROM
    KONRAD.STUDENT, TEACHER
WHERE
    CLASS1_TEACHER = TEACHER_ID
```

通过在语句中使用限定的表名 KONRAD.STUDENT 代表 STUDENT，可告诉 DBMS 从 Konrad 拥有的 STUDENT 表中提取数据，而不是试图从不存在的用户所创建（或拥有）的 STUDENT 表中获得数据。

技巧 8 理解视图

视图是虚拟的表。视图看起来像表，因为它具有表的所有实质性的组成——有名称、有以命名列排列的数据行，还有与所有其他真正的表一起保存在数据库目录中的定义。另外，可在许多可使用表名的 SQL 语句中使用视图名。让视图成为“虚拟”表而不是“真正”表的原因在于，在视图中看到的数据存在于用于创建视图的表中，而不存在于视图本身。

为了创建一个视图，可使用 SQL 的 CREATE VIEW 语句。例如，假设有一个图 1.4 所示的数据库表，其中有销售人员及其工资的数据。

销售代表表

EMP_NUM	NAME	APPT_COUNT	SALES_COUNT	SSAN	ADDR
1	Tamika James	10	6		
2	Sally Wells	23	9		
3	Robert Hardy	17	12		
4	Jane Smith	12	8		
5	Rodger Dodger	22	17		
6	Cade Wilans	18	10		

工资表

EMP_NUM	YTD_SALARY	YTD_COMMISSION
1	\$69,565.00	\$2,565.00
2	\$89,498.00	\$16,323.00
3	\$43,000.00	\$27,123.00
4	\$75,000.00	\$17,000.00
5	\$60,000.00	\$5,000.00
6	\$72,898.00	\$2,993.00

图 1.4 用于视图的数据库表的例子

当执行以下 SQL 语句时，DBMS 将视图的定义保存在名为 APPT_SALES_PAY 的数据库中。与创建实际空白表的 CREATE TABLE 语句不同，CREATE TABLE 语句在系统目录中保存表，而 CREATE VIEW 语句只保存视图的定义：

```

CREATE VIEW APPT_SALES_PAY
  (NAME, APPTS, SALES, SALES_PCT, YTD_SALARY, YTD_COMMISSION) AS
SELECT
  NAME, APPT_COUNT, SALES_COUNT, ((APPT_COUNT / SALES_COUNT)
    * 100), YTD_SALARY, YTD_COMMISSION
FROM
  SALES_REPS, PAYROLL
WHERE
  SALES_REPS.EMP_NUM = PAYROLL.EMP_NUM

```

在创建了视图之后，就可以将其用在 SELECT 语句中，就好像它是真正的表一样。例如，在创建了 APPT_SALES_PAY 视图之后，通过使用以下 SQL 语句，可显示定义视图的查询结果：

```
SELECT * FROM APPT_SALES_PAY
```

对于上面的 CREATE VIEW 语句来说，DBMS 将执行多表选择语句（读者将在技巧 152 “使用带 FROM 子句的 SELECT 语句进行多表查询”中学习更多知识）来形成如图 1.5 所示的虚拟表，然后在虚拟表的每一行中的所有列上显示值。

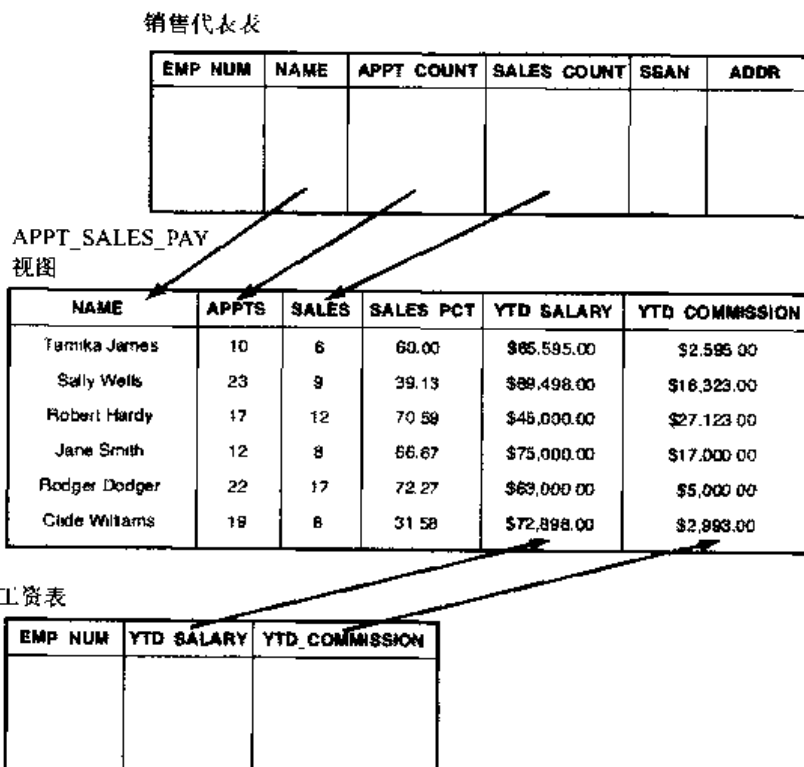


图 1.5 根据基表 SALES_REPS 和 PAYROLL 生成的 APPT_SALES_PAY 视图

有时，DBMS 需要执行视图所定义的查询并将结果保存在临时表中。然后 DBMS 使用临时表来执行引用视图的 SQL 语句（如 SELECT * FROM APPT_SALES_PAY）。当 DBMS 不再需要临时表时（即在完成 SQL 语句之后），DBMS 就将临时表抛弃。请记住，视图并不保存数据，它们只是显示保存在基表（在当前的例子中是 SALES_REPS 和 PAYROLL 表）中的数据。

不管 DBMS 是通过快速地创建行来处理视图还是通过将视图数据放入临时表，最终结果是一样的——用户可在 SQL 语句中引用视图，就好像数据库中真正的表一样。

使用视图有以下好处：

- 提供安全性。当不希望用户看到表中的所有数据时，就可以使用视图让用户只看到指定的列。
- 简化了数据结构。可将数据库以“个性化”的表集呈现出来。
- 抽象的数据结构。随着时间的推移，某些用户将保存常常使用的 SQL 查询，而另一些人甚至可能编写从数据库中提取数据并生成报告的 Visual Basic 或 C++ 程序。例如，如果某人（如表的拥有者或数据库管理员）通过将一个表分成两个表而改变了表的物理结构，保存的用户查询可能不再好用，而应用程序则试图访问“虚拟的”视图表中的数据，此时可将视图与对底层数据库的变化隔离开来。例如，当将表分开时，需要改变视图查询，使其重新将分开的表组合成原来视图中的列集。
- 简化查询。通过使用一个视图将几个表中的数据组合成单个的虚拟表，这就使得用户根据单个表来编写 SQL 查询成为可能，这可避免使用多表 SELECT 和 JOIN 语句的复杂性。

使用视图有两个主要缺点：

- 性能。由于视图是虚拟的表，在使用包括视图引用的 SQL 语句时，除了执行所键入的 SQL 语句中的查询或更新之外，还要告诉 DBMS 执行定义视图的查询。
- 更新限制。遗憾的是，SQL 违反了 Codd 规则中的第 6 条规则（在技巧 4 “理解 Codd 的 12 条关系数据库定义规则”中所学习的），因为并非所有视图都是可更新的。目前，SQL 将可更新的视图限制为基于对单个表的没有 GROUP BY 或 HAVING 子句的查询。除此之外，为了使视图是可更新的，视图不能有总计函数、计算的列或是 SELECT DISTINCT 子句。最后，视图必须包括表的键列，而且任何排除在视图之外的列在基表中必须可以为空。

由于 SQL 对可用于更新基表视图的限制，用户不能总是用创建视图来代替基表。另外，在使用视图的情况下，请估计一下使用视图的好处与让 DBMS 在每次执行引用视图的 SQL 语句时创建虚拟表所引起的性能方面的损失。

技巧 9 理解架构

表由处理特定实体类型的数据的行和列组成，如营销电话、销售统计数字、顾客、定单、工资等。架构是相关表的集合。因而架构对于表来说就像是表对单独的数据项一样。

假设用户正在为如图 1.6 所示的有 5 个部门的销售机构工作。

架构可包括：

- 表 (Tables)。以行列排列的描述了物理或逻辑实体的相关数据项。架构包括表以及表的所有组成（在技巧 5 “理解表”中描述过）。
- 视图 (Views)。显示来自一个或多个基表中的数据的 SQL 查询所定义的虚拟表（如在技巧 8 “理解视图”中所描述的）。架构包括使用在架构内包括实际数据的基表的所有视图定义。
- 断言 (Assertions)。对架构中表之间的数据关系施加限制的数据库完整性约束。读者将在技巧 18 “理解断言”中学习有关断言的知识。
- 权限 (Privileges)。单独用户或用户组必须创建或修改表结构或查询和更新数据库表中数据（或是通过视图指定的列中的数据）的访问权限。读者将在技巧 98~119 中学

习有关安全性权限方面的知识。

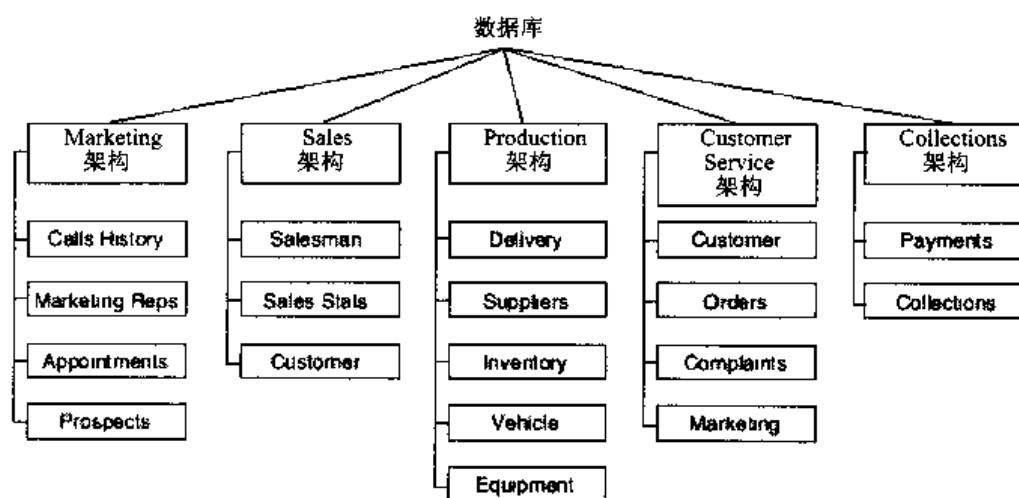


图 1.6 将表组合成 5 种架构的数据库，每种架构用于公司中的一个部门

- 字符集 (Character sets)。用于允许 SQL 显示非罗马字符，如 Cyrillic (俄罗斯)、Kanji (亚洲) 等的数据库结构。
- 校订 (Collations)。为字符集定义排列顺序。
- 翻译 (Translations)。控制从一种字符集翻译为另一种字符集时的文本字符顺序。例如，翻译允许以 Kanji 字符集保存数据而以 Cyrillic、Kanji 和 Roman 显示数据——根据用户数据的视图而定。翻译除了表明在一个字符集中的字符映射为另一字符集中的哪些字符之外，还定义当用于比较操作时，一种字符集中的文本字符串如何与另一字符集中的文本字符串加以比较。

简短地说，架构是保存表集、描述表中数据（列）的元数据、域和限制表中列中的数据的约束、限制可向表中添加或从表中删除的行的键（主键和外键），以及定义对架构中的对象能做什么的安全性的容器。

当使用 CREATE TABLE 语句（读者将在技巧 29 “使用 CREATE TABLE 语句创建表”中学习有关知识）时，DBMS 自动地以默认的架构（即名为<user ID>的架构）和交互式会话方式创建表。这样一来，如果用户 Konrad 和 Karen 每人都登录到数据库并执行以下 SQL 语句，DBMS 将向两个架构中的每一个中都添加一个表，如图 1.7 所示。

```

CREATE TABLE CALL_HISTORY
(PHONE_REP_ID CHAR(3) NOT NULL,
PHONE_NUMBER INTEGER NOT NULL,
DATE_TO_CALL DATE,
TIME_TO_CALL INTEGER,
DATE_CALLED DATE NOT NULL,
TIME_CALLED INTEGER NOT NULL,
HANGUP_TIME INTEGER NOT NULL,
DISPOSITION CHAR(4) NOT NULL)
  
```

简短地说，任何时候使用 CREATE 语句来创建诸如表、视图、域、断言等对象，DBMS 都将在默认的“容器”架构中创建对象。

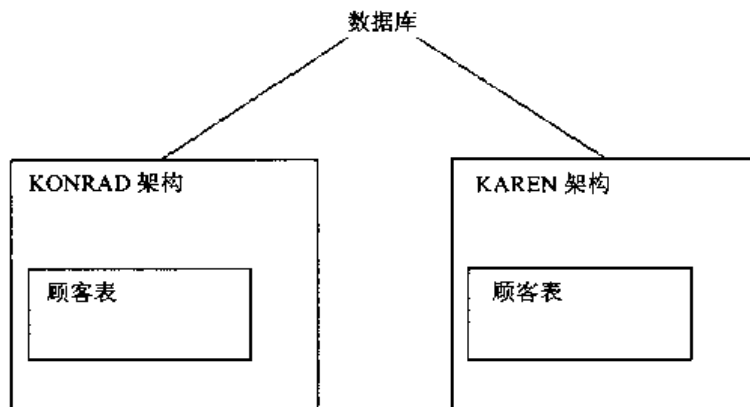


图 1.7 带有两个架构 KONRAD 和 KAREN 的数据库

如果希望在特定的架构“容器”内创建对象，容器必须存在，而且对要创建的对象必须使用限定的名称。限定的对象名称是在技巧 6 “理解表名”中学过的限定表名的扩展。不过不是使用<owner>.<object name>，而是使用<schema name>.<object name>在特定的架构内放置对象。

例如，为了创建图 1.6 中所示的 MARKETING 架构，可使用以下 SQL 语句创建 MARKETING 架构及其 4 个表的结构：

```

CREATE SCHEMA MARKETING AUTHORIZATION KONRAD
CREATE TABLE CALLS_HISTORY
  (PHONE_REP_ID CHAR (3) NOT NULL,
   PHONE_NUMBER INTEGER, NOT NULL,
   DATE_CALLED DATE)
CREATE TABLE MARKETING REPS
  (REP_ID CHAR(3),
   REP_NAME CHAR(25))
CREATE TABLE APPOINTMENTS
  (APPOINTMENT_DATE DATE,
   APPOINTMENT_TIME INTEGER,
   PHONE_NUMBER INTEGER)
CREATE TABLE PROSPECTS
  (PHONE_NUMBER INTEGER,
   NAME CHAR(25),
   ADDRESS CHAR(35))
  
```

在 CREATE SCHEMA 语句中的 AUTHORIZATION 授权 Konrad 可修改架构及其对象。因此，Konrad 就可以使用 ALTER TABLE MARKETING.<table name>语句来改变 MARKETING 架构中包括的表中的列、域和对列的约束。另外，Konrad 可在 MARKETING 架构中通过在 CREATE TABLE 语句中指定架构名来创建额外的表，例如：

```

CREATE TABLE MARKETING.CONTESTS
  (DESCRIPTION CHAR(25),
   RULES VARCHAR(100),
   WIN_LEVEL1 MONEY,
   WIN_LEVEL2 MONEY,
   WIN_LEVEL3 MONEY)
  
```

所有的 DBMS 产品都有保存表的集合和相关对象的架构或“容器”。但是，架构的名称却

随产品不同而有所不同。例如 Oracle、Informix 和 Sybase 要求架构名和用户名是相同的。这每一种产品还限制了在 CREATE SCHEMA 语句中可定义的对象类型。这样一来，用户必须检查 DBMS 手册（或帮助系统）中 CREATE SCHEMA 语句的句法，看一看可在架构“容器”将什么样的对象组合在一起以及可以为架构取什么样的名称。

技巧 10 理解域

域是表中特定列中合法值的集合。例如，假设 EMPLOYEE 表中有 DEPENDANT 字段按公司策略要求必须是 0 和 14 间的整数。那么 DEPENDANT 的域就是 0、1、2、3、4、5、6、7、8、9、10、12、13、14。

当使用 CREATE DOMAIN 语句（此内容将在技巧 128 “使用 CREATE DOMAIN 语句创建域”中讨论）定义了域之后，在定义列时，用户可将域用作数据类型。例如，假设有一个 CUSTOMER 表，其中有 STATE 字段。通过使用以下 SQL 语句创建一个 STATE_CODE 域来定义 STATE 字段的域：

```
CREATE DOMAIN STATE_CODE AS CHAR(2)
    CONSTRAINT VALID_STATE_ABBREVIATION
    CHECK (VALUE IN ('AL', 'AK', 'AZ', 'CO', 'CT', ... ))
```

注意：可用其余的 45 个州的代码来代替 CREATE DOMAIN 语句中 VALUE IN 部分中的“...”。

当创建表时，在向 CUSTOMER 表中键入 STATE 字段的值时，可使用 STATE_CODE 域作为该字段的数据类型，从而让 DBMS 确认键入的数据，如以下 SQL 语句所示：

```
CREATE TABLE CUSTOMER
    (NAME VARCHAR(25),
    ADDRESS VARCHAR(35),
    CITY VARCHAR(20),
    STATE STATE_CODE,
    ZIP_CODE INTEGER)
```

定义域的美妙之处在于，可快速地改变域而不必改变表的结构或是重新编译任何已有的存储过程或应用程序。

例如，假设 Puerto Rico（波多黎哥）要成为一个州，可使用 ALTER DOMAIN 语句将 PR 添加到合法的州简写的清单中。DBMS 然后就自动地允许用户在 STATE 字段内键入 PR，因为更新的 STATE_CODE 域（保存在系统表中）将把 PR 包括在可添加到 CUSTOMER 表中作为一行的列值中。

技巧 11 理解约束

约束是限制用户或应用程序键入表列数据的数据库对象。有 7 种类型的约束：断言（assertions）、域（domains）、检查约束（check constraints）、外键约束（foreign key constraints）、主键约束（primary key constraints）、必需数据（required data）和惟一性约束（uniqueness constraints）。每种约束类型都在维护数据库的完整性上起着不同的作用。

- 断言。允许用户在数据库内的多个表间维护数据值间关系的完整性。读者将在技巧 18 “理解断言”和技巧 151 “使用 CREATE ASSERTION 语句创建多表约束”中学习

有关知识。

- 域。保证用户和应用程序向表列中只能键入合法的值。读者在技巧 10“理解域”中已学习了有关知识。
- 检查约束。除了用于定义域和声明约束之外,这个约束可直接应用于 CREATE TABLE 或 ALTER TABLE 语句的表列中。不管检查约束是取了名的(使用 CREATE DOMAIN 或 CREATE ASSERTION 语句),还是直接添加到表的定义中的,都执行同样的功能。

可通过对数据值的不变集合为检查约束起一个名称来创建域。不用使用 CREATE DOMAIN 语句,可将检查约束(在技巧 145“使用 CHECK 约束确认列值”中将学习有关知识)直接包括到 CREATE TABLE 或 ALTER TABLE 语句中的一列中。

断言实际上是使用 CREATE ASSERTION 语句向 CHECK 约束赋予的另一个名称。可以使用断言或是多表 CHECK 约束把商业规则应用于表中的列值。例如,假设公司不允许返回定单。因此,可在 CHECK 约束中对 ORDER 表的 QUANTITY 列使用查询,这将只允许小于当前库存(如 INVENTORY 表中所表明的)的产品总数的值。读者将在技巧 345“理解何时用 CHECK 约束代替触发器”中学习有关使用搜索条件方面的内容。

- 外键约束。在数据库内确保在有子记录的情况下父记录不能被删除,维护引用完整性。反之, FOREIGN KEY 约束确保如果没有相应的父记录就不能向表中添加子记录(行)。
- 主键约束。通过指定表的每行中至少有一列必须具有惟一值来维护实体完整性。使表中每行中的列具有不同的值可防止表的两行一样,从而满足了 Codd 的第 2 条规则(保证访问规则,已在技巧 4“理解 Codd 的 12 条关系数据库定义规则”中讨论了这条规则)。例如,如果有一个 STUDENT 表,用户希望表中的一行且只有一行列出任意学生的属性(列)。那么,可向 STUDENT 表的 STUDENT_ID 列应用 PRIMARY KEY 约束,以保证没有两个学生具有相同的学生 ID 号。
- 必需数据。表中的某些列必须包含数据,以便使行能成功地描述一个物理或逻辑实体。例如,假设有一个 GRADES 表,其中包括一个名为 STUDENT_ID 的列。表中的每行必须在 STUDENT_ID 列中有一个值,从而使成绩记录(行)有意义。读者将在技巧 143“使用 NOT NULL 列约束防止列中的 NULL 值”中学习有关 NOT NULL(必需数据)约束的知识。
- 惟一性约束。虽然每个表可以只有一个 PRIMARY KEY 值,但有时也想要指定表中每一行的不止一列上具有惟一值。这时可以对表列使用 UNIQUE 约束(读者将在技巧 144“使用 UNIQUE 列约束防止列中的重复值”中学习有关知识),从而保证表中只有一行在该列中有某一值。

当约束(CHECK、FOREIGN KEY、PRIMARY KEY、NOT NULL [必需数据]、UNIQUE)通常作为表定义的一部分而指定或是使用 CREATE 语句(ASSERTION、DOMAIN)时,DBMS 在其系统表中保存每个约束的描述。所有约束都是数据库对象,既可限制向表列中输入的值,也可限制可向表中添加的行。

技巧 12 理解数据定义语言(DDL)

数据定义语言(DDL)是一套 SQL 语句(ALTER、CREATE、DROP、GRANT),使用户

可以创建、修改或是删除组成关系数据库的对象。换一句话说，用户可以用 DDL 来定义数据库的结构和安全性。SQL-89（为 SQL 编写的第一个 ANSI/ISO 标准）将数据操纵语言（DML）和 DDL 定义为截然不同而相对没有关系的语句。

最基本（且功能强大）的 DDL 语句是 CREATE 语句。使用 CREATE 可构建数据库架构（已经在技巧 9“理解架构”中学过）。例如，为了构建带有两种架构的如图 1.8 所示的数据库，可使用以下 SQL 语句：

```
CREATE SCHEMA AUTHORIZATION KONRAD
CREATE TABLE EMPLOYEES
    (ID CHAR(3),
     NAME VARCHAR(35),
     ADDRESS VARCHAR(45),
     PHONE_NUMBER CHAR(11),
     DEPARTMENT CHAR(10),
     SALARY MONEY,
     HOURLY_RATE MONEY)
CREATE CUSTOMERS
    (NAME VARCHAR(35),
     ADDRESS VARCHAR(45),
     PHONE_NUMBER CHAR(11),
     FOOD_PLAN CHAR(2))
CREATE TABLE APPT_SCHEDULE
    (APPT_DATE DATE,
     APPT_TIME INTEGER,
     APPT_DISPOSITION CHAR(4),
     APPT_SALESMAN_ID CHAR(3))
GRANT SELECT, UPDATE
    ON EMPLOYEES
    TO HR_DEPARTMENT
GRANT ALL PRIVILEGES
    ON CUSTOMERS
    TO MARKETING_REPS, OFFICE_CLERKS
GRANT SELECT
    ON APPT_SCHEDULE
    TO PUBLIC
GRANT SELECT, INSERT
    ON APPT_SCHEDULE
    TO MARKETING_REPS
CREATE SCHEMA AUTHORIZATION KAREN
CREATE TABLE EMPLOYEES
    (ID CHAR(3),
     NAME VARCHAR(35),
     ADDRESS VARCHAR(45),
     PHONE_NUMBER CHAR(11),
     EMPLOYEE_TYPE CHAR(2),
     SALARY MONEY,
     HOURLY_RATE, MONEY)
GRANT SELECT, UPDATE
    ON EMPLOYEES
```

```

TO HR_DEPARTMENT
CREATE PATIENTS
  (ID INTEGER
   SPONSOR_SSAN CHAR(11),
   NAME VARCHAR(35),
   ADDRESS VARCHAR(45),
   PHONE_NUMBER CHAR(11),
   AILMENT_CODES VARCHAR(120))
GRANT SELECT, UPDATE
  ON PATIENTS
  TO DOCTORS, NURSES, CLERKS
CREATE TABLE APPT_SCHEDULE
  (APPT_DATE DATE,
   APPT_TIME INTEGER,
   REASON_CODES VARCHAR(120),
   DOCTOR_ID CHAR(3),
   NURSE_ID CHAR(3),
   REFERRAL_DOCTOR_ID CHAR(15))
GRANT SELECT, UPDATE
  ON APPT_SCHEDULE
  TO PUBLIC

```

注意：CREATE SCHEMA 语句在命名架构时使用 AUTHORIZATION 代替 USER（如在本例中的 CREATE SCHEMA AUTHORIZATION KONRAD）。SQL-89 规范不仅把用户看作是授权 ID，而且术语授权 ID 在为部门而不是个人创建架构时更为适用。另外，即使在为个人授权 ID “工作” 创建架构时，用户 ID（或用户名）就是被授予对架构中的对象拥有权的 ID。

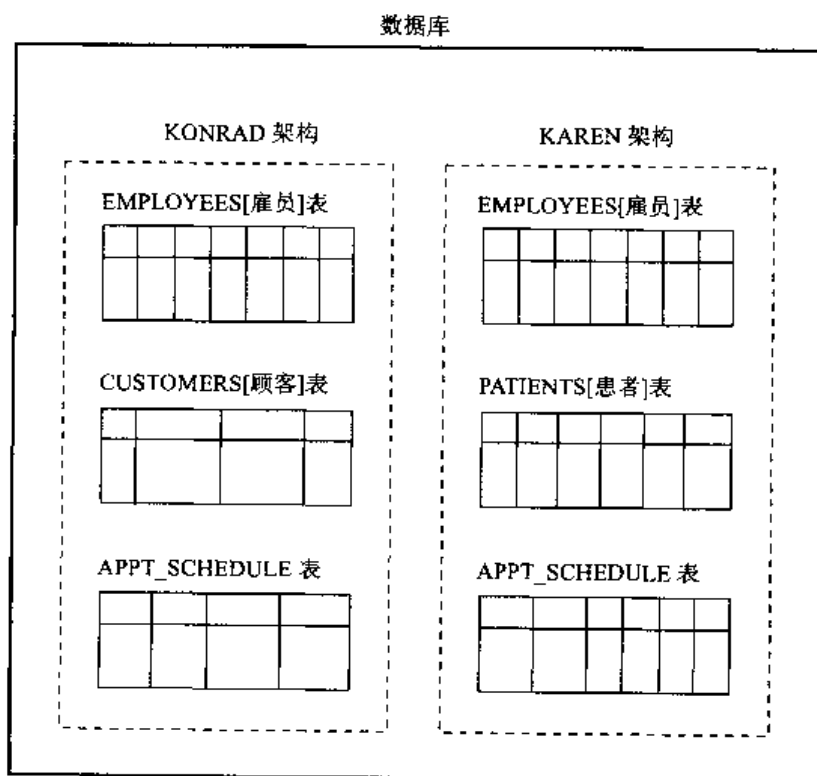


图 1.8 带有名为 KONRAD 和 KAREN 的两种架构，每种都包括 3 个表的示例数据库

虽然在本技巧中只展示了 CREATE SCHEMA 语句,但在本书其他技巧中将要展示如何使用以下的每条 DDL 语句:

- CREATE/DROP/ALTER ASSERTION。根据单个或多个表列关系限制可赋予列的值。DDL 断言语句将在技巧 151 “使用 CREATE ASSERTION 语句创建多表约束”中详细地讨论。
- CREATE/DROP/ALTER DOMAIN。一组赋予某列的合法值。DDL 域语句将在技巧 128 “使用 CREATE DOMAIN 语句创建域”中讨论。
- CREATE/DROP INDEX。可加速数据库访问的结构,通过这种结构可使 SQL 查询语句更容易找到具有满足搜索标准的列的行。DDL 索引语句将在技巧 121 “使用 CREATE INDEX 语句创建索引”和技巧 123 “使用 MS-SQL Server Enterprise Manager 创建索引”中加以讨论。
- CREATE/DROP SCHEMA。一套有关系的表、视图、域、约束和安全性结构。此内容将在技巧 399 “使用 CREATE SCHEMA 语句创建表并授予对此表的访问权限”中讨论。
- CREATE/DROP/ALTER TABLE。有系列(数据)的行。DDL 表语句将在技巧 29 “使用 CREATE TABLE 语句创建表”、技巧 42 “使用 ALTER TABLE 语句改变主键和外键”中讨论。
- CREATE/DROP/ALTER VIEW。显示一个或多个基表行中的数据列的虚拟表。DDL 视图语句将在技巧 153 “使用视图显示一个或多个表或视图中的列”以及技巧 355 “使用 ALTER VIEW 语句修改视图”中加以讨论。
- GRANT。向用户或用户组授予对单独数据库对象的特定的 SQL 语句访问权。GRANT 语句将在技巧 106 “使用 GRANT 语句的 WITH GRANT OPTION 允许用户向其他用户授予对数据库对象的访问权”中加以讨论。

现在要了解的重要事情是,DDL 是由让人们创建、修改并删除数据库中的对象(表、视图、索引、域、约束)的语句所组成的。DDL 还有可用于建立数据库安全性的 GRANT 语句,这是通过一条语句一条语句地或一个对象一个对象地授予用户或用户组以 DDL 和 DML 语句访问权而实现的。

注意:GRANT 语句是 DDL 和数据控制语言(DCL)的一部分。当在 CREATE SCHEMA 语句中使用时,GRANT 的作用是 DDL 语句。当用于在架构定义外给予用户(或用户组)额外的权限时,其作用是 DCL 语句。因此,读者将在技巧 14 “理解数据控制语言(DCL)”中找到 GRANT 及其对立面(REVOKE),其中讨论了 SQL 的 DCL 语句。

技巧 13 理解数据操纵语言(DML)

数据操纵语言(DML)可让用户对 SQL 数据库做 5 件事情:向表中添加数据、提取并显示表列中的数据、修改表中的数据以及从表中删除数据。由此,基本的 DML 由以下 5 条语句组成:

- INSERT INTO。让用户向表中添加一个或多个行(或列)。
- SELECT。让用户查询一个或多个表并显示满足搜索条件的行中的列。
- UPDATE。让用户改变表行中满足搜索条件的一个或多个列中的值。

- **DELETE FROM**。让用户删除满足搜索条件的一个或多个表行。
- **TRUNCATE**。让用户从表中删除所有的行。

在理论上，数据操纵是很简单的。前面已经理解了添加数据的意义。数据操纵较难的部分是选择想要显示的行、加以修改或删除。读者可以阅读后续章节了解这些语句的用法。

虽然基本的 DML 只由 5 条语句组成，但它是对数据库输入、显示、修改以及删除数据的功能强大的工具。DML 让用户准确指定想要对数据库中的数据做什么。

技巧 14 理解数据控制语言（DCL）

虽然 DML 可让用户对数据库中的数据作出改变，但数据控制语言（DCL）却保护数据不受损害。如果正确地使用 DCL 提供的工具，就可使未授权用户不能查看或是改变数据并防止许多可损害数据的问题发生。共有 4 种 DCL 命令：

- **COMMIT**。通过更新永久数据库表以与更新了的临时副本匹配，从而告诉 DBMS 将临时副本变为永久的改变。COMMIT 语句将在技巧 94 “理解何时使用 COMMIT 语句”中加以讨论。
- **ROLLBACK**。在最近的提交之后，告诉 DBMS 取消对 DBMS 所做的任何变化。ROLLBACK 语句将在技巧 95 “使用 ROLLBACK 语句取消对数据库对象所做的改变”中加以讨论。
- **GRANT**。向用户或用户组授予对单独的数据库对象的特定 SQL 语句访问权。GRANT 语句将在技巧 106 “使用 GRANT 语句的 WITH GRANT OPTION 允许用户向其他用户授予对数据库对象的访问权”中加以讨论。
- **REVOKE**。从用户或用户组中撤消以前授予的对单独的数据库对象的特定的 SQL 语句访问权。REVOKE 语句将在技巧 108 “使用带 CASCADE 选项的 REVOKE 语句删除权限”中加以讨论。

数据库在某人改变它的时候是最易受损害的。如果在改变的中间，软件或硬件出了故障，则数据将留在中间状态——部分想要的改变已经发生，而部分却未发生。例如，假设告诉 SQL 将资金从一个银行账户移动到另一银行账户。如果正在转移时计算机死锁，用户将无法知道 DBMS 是否已经从一个账户中提出了资金或资金是否已经转到了另一账户。

通过在事务内封装借方和贷方的 UPDATE 语句，可确保 DBMS 在执行 COMMIT 语句将更新的余额永久地写入数据库之前成功地执行那两个语句（即借和贷的 UPDATE 语句）。

如果 DBMS 不能成功地在事务内完成语句，可发出 ROLLBACK 命令。DBMS 将返回最近的 COMMIT 命令执行后的原来状态。在资金转移失败的例子中，DBMS 将返回原来状态，从而使账户中的余额返回 DBMS 试图将资金从一个账户移动到另一账户之前的状态，就好像该事务从未发生过一样。

除了由硬件或软件引起的数据损坏之外，还必须保护数据不受用户本身带来的损害。其他人应该能够看到某些但不是全部的数据，但却不能更新任何数据。而且其他人应该有查看的访问权以及更新部分数据的访问权。这样一来，必须能够逐用户以及按用户组实施数据库的安全性。DCL 给出了 GRANT 和 REVOKE 命令，可向单独用户或用户组赋予访问权限。用于控制安全性的 DCL 命令有：

- **GRANT SELECT**。可让用户或用户组查看表或视图中的数据。技巧 110 讨论了

GRANT 和 REVOKE SELECT 语句。

- REVOKE SELECT。防止用户或用户组查看表或视图中的数据。
- GRANT INSERT。可让用户或用户组向表或视图中添加行。
- REVOKE INSERT。防止用户或用户组向表或视图中添加行。
- GRANT UPDATE。可让用户或用户组改变表或视图中的列值。技巧 113 讨论了 GRANT UPDATE 语句。
- REVOKE UPDATE。防止用户或用户组改变表或视图中的列值。
- GRANT DELETE。允许用户或用户组删除表或视图中的行。
- REVOKE DELETE。防止用户或用户组删除表或视图中的行。
- GRANT REFERENCES。可让用户或用户组定义对表的 FOREIGN KEY 引用。技巧 114 讨论了 GRANT REFERENCES 语句。
- REVOKE REFERENCES REVOKE REFERENCES。防止用户或用户组定义对表的 FOREIGN KEY 引用。

这样一来，DCL 包括可用于控制谁可访问数据库以及哪些用户在登录到数据库可做什么的命令。另外，DCL 让用户控制何时 DBMS 对数据库做出永久的改变（COMMIT）以及允许用户取消（ROLLBACK）还没有提交的改变。

技巧 15 理解标准 SQL 的日期时间数据类型和 DATETIME 数据类型

虽然可在 CHAR 或 VARCHAR 的列中保存日期和时间，但使用日期时间列将更为方便。如果在日期时间列中放入日期和时间，DBMS 在作为 SELECT 语句的一部分显示日期时间列的内容时将把日期和时间以标准的方式格式化。更为重要的是，使用日期时间列，能够使用专门的日期和时间函数（如 INTERVAL 和 EXTRACT）来操作日期和时间数据。

SQL-92 标准指定 5 种日期时间数据类型：

- DATE。使用 10 个字符保存四位数的年、二位数的月和二位数的日的日期值，其格式为 2000-04-25。由于 DATE 数据类型使用四位数的年，可使用它来代表任何从 0001 年到 9999 年的任何日期。因而 SQL 将会遇到 10K 年的问题（与计算机中的 2000 年问题类似——译者注），对于笔者这样的人来说，这种问题只能让将来的后代去操心了。
- TIME。使用 8 个字符，包括逗号，来表示两位数的小时、两位数的分和两位数的秒，其格式为 19:22:34。由于 SQL 使用 24 小时的时钟来格式化时间，所以 19:22:34 表示下午 7 时 22 分 34 秒，而 07:22:34 表示上午 7 时 22 分 34 秒。如果将一列定义为 TIME 类型，在默认情况下 DBMS 只显示整秒数。但是，可在定义列为 TIME 类型时向 TIME 类型添加精度，从而让 DBMS 存储（显示）秒的小数部分。例如，如果使用以下 SQL 语句来创建表的话：

```
CREATE TABLE time_table  
  (time_with_seconds TIME(3))
```

DBMS 将把时间数据存储为包括多达 3 位的数用来代表千分之几秒。

- TIMESTAMP。既包括日期也包括时间，使用 26 个字符——10 个字符用于保存日期，后跟一空格（用于分隔目的），然后用 15 个字符显示时间，包括默认的 6 个小数位

的小于 1 秒部分。这样一来，如果使用以下语句创建一个表：

```
CREATE TABLE time_table
(timestamp_column TIMESTAMP,
timestamp_column_no_decimal TIMESTAMP (0))
```

DBMS 将在格式化为 2000-04-25 19:22:34.123456 的 `TIMESTAMP_COLUMN` 列中保存日期和时间，而在 `TIMESTAMP_COLUMN_NO_DECIMAL` 列中以格式 2000-04-25 19:25:34 保存日期和时间（在 `TIMESTAMP` 之后的括号（0）中的数字指定秒部分的精度，在本例中该数字为 0）。

- **TIME WITH TIME ZONE**。使用 14 个字符来表示日期和时间以及距离 Universal Coordinated Time (UTC，通用协调时间) 的偏移——8 个字符用于保存时间后跟本地时间与 (UTC) ——从前称为格林威治时间或是 GMT——的偏移。

因而，如果使用以下语句创建一个表：

```
CREATE TABLE time_table
(time_with_gmt TIME WITH TIME ZONE,
time_with_seconds_gmt TIME (4) WITH TIME ZONE)
```

DBMS 将在 `TIME_WITH_GMT` 列中以格式 19:22:24-05:00 保存时间，而在 `TIME_WITH_SECONDS_GMT` 列中以格式 19:22:24.1234-05:00（在 `TIME_WITH_SECONDS_GMT` 列的数据类型中的(4)表示可指定用来代表时间中秒的小数部分的可选的精度）。

- **TIMESTAMP WITH TIME ZONE**。使用 32 个字符来表示日期、时间和与 Universal Coordinated Time (UTC) 的偏移——10 个字符保存日期，后跟一空格用于分隔目的，然后是 21 个字符表示时间，包括默认的 6 位秒的小数部分以及与 UTC (GMT) 的偏移。这样一来，如果使用以下语句创建一个表：

```
CREATE TABLE time_table
(timestamp_column TIMESTAMP WITH TIME ZONE,
timestamp_no_dec TIMESTAMP(0)WITH TIME ZONE)
```

DBMS 在 `TIMESTAMP_COLUMN` 列中以格式 19:22:34.123456+04:00 保存日期和时间，而在 `TIMESTAMP_NO_DEC` 列中使用格式 2000-04-25 19:25:34+01:00 来保存日期和时间（`TIMESTAMP` 后面括号（0）中的数字指定时间的秒部分的小数位的精度，在本例中是 0）。

遗憾的是，并不是所有的 DBMS 产品都支持所有这 5 种标准的 SQL 日期时间数据类型。事实上，某些 DBMS 产品甚至将 `TIMESTAMP` 用作与定义保存日期和时间数据列不同的目的。因而，请检查一下自己的系统手册，查看一下自己的 DBMS 到底支持哪种 SQL 的日期时间数据类型。

如果发现系统使用非标准的数据类型，如 `DATETIME`（由 SQLBase、Sybase 和 MS-SQL Server 所使用的）也不要惊奇。

如果系统中使用了 `DATETIME` 数据类型，可使用类似于以下的 SQL 语句定义一系列用来保存日期和时间：

```
CREATE TABLE date_table
(date_time DATETIME)
```

为了向 `DATETIME` 列插入日期和时间，可将日期和时间括在单括号内，使用类似于以下的 `INSERT` 语句：

```
INSERT INTO date_table
VALUES ('04/25/2000 21:05:06:123')
```


如果正在使用 MS-SQL Server 且执行以下 SQL 语句:

```
SELECT * FROM date_table
```

DBMS 将把 DATE_TIME 列中的值显示为 2000-04-25 21:05:06.123。

MS-SQL Server 允许用户在 INSERT 语句中使用多种格式的变种之一指定日期,其中包括但不限于以下各种:

- Apr 25 2000
- APR 25 2000
- April 25, 2000
- 25 April 2000
- 2000 April 25
- 4/25/00
- 4-25-2000
- 4.25.2000

MS-SQL Server 还为用户提供许多方式来表达想要插入到 DATETIME 列中的时间。表示时间的合法方式包括:

- 9:05:06:123pm
- 9:5:6:123pm
- 9:05pm
- 21:00
- 9pm
- 9PM
- 9:05

(注意,在本例中最后的条目[“9:05”]插入的是上午 9:05 而不是下午 9:05)。如果插入日期而没有时间,MS-SQL Server 将向日期中追加 00:00:00.000。这样一来,以下 SQL 语句将把 DATE_TIME 值设置为 2000-04-10 00:00:00.000(MS-SQL Server 将用 0 来代替空下来的时间部分)。

```
INSERT INTO date_table VALUES ("2000 Apr 25")
```

如果只向 DATETIME 插入时间,MS-SQL Server 将用 01/01/1900 代替所忽略的日期。

技巧 16 理解 SQL 的 BIT 数据类型

当处理那种只有两个值的数据时,可使用 BIT 数据类型。例如,可使用 BIT 字段来存储 yes/no (是/否) 或 true/false (真/假) 调查问题的答案,如 Are you a homeowner? (你有房屋吗?)

可使用字母 Y、N、T 和 F 在 CHARACTER 列中存储对 yes/no 和 true/false 问题的回答。每个数据值将占用 1 个字节(8 位)的存储空间。如果使用 BIT 列,则可使用 1/8 的空间存储同样数量的数据。

例如,假设使用以下 SQL 语句创建了 CUSTOMER 表:

```
CREATE TABLE customer  
(id INTEGER,  
 name VARCHAR(25),  
 high_school_graduate BIT,  
 some_college BIT,
```

```
graduate_school BIT,  
post_graduate_work BIT,  
male BIT,  
married BIT,  
homeowner BIT,  
US_citizen BIT)
```

如果遵循通常的约定，在 BIT 列中的 1 将表示 TRUE，而 0 则表示 FALSE。这样一来，如果 MARRIED 列的值为 1，则意味着该 CUSTOMER 是已婚的。类似地，如果 US_CITIZEN 列中的值为 0，则意味着该 CUSTOMER 不是美国公民。

在本例中对于 8 个两态（BIT）的列使用 BIT 数据类型而不是 CHARACTER 数据类型，仅每行就节约了 56 字节的存储空间，而且还简化了基于两态列值的查询。

例如，假设想要获得所有男性顾客的清单。如果 MALE（男）列是 CHARACTER 类型，则必须了解该列是包括 T、t、Y、y 还是其他用于指出 CUSTOMER 是男性的值。当该列为 BIT 列时，人们知道该列的值只能是 1 或 0——如果该 CUSTOMER 是男性，则很可能是 1，因为根据约定，1 表明 TRUE。

可使用 BIT 列通过检查 SQL 语句中 WHERE 子句中的列值来选择满足特定条件的行。例如，可使用以下 SQL SELECT 语句来制作所有从高中毕业的顾客的清单：

```
SELECT id, name  
FROM customer  
WHERE high_school_graduate = 1
```

选择满足几个条件之一的行也是容易的。例如，假设希望得到已婚的或是拥有房产的顾客清单。可使用以下 SQL SELECT 语句达到目的：

```
SELECT id, name  
FROM customer  
WHERE married = 1 OR homeowner = 1
```

另一方面，如果想要只选择已婚的有房产的顾客，可使用 AND 代替上面 WHERE 子句中的 OR。

技巧 17 理解 MS-SQL Server 的 IDENTITY 属性

可向表中的一列（只能是一列）应用 IDENTITY 属性，让 MS-SQL Server 为所添加的行的未指定列值的某列提供一个增加的、非 NULL 值。例如，假设想要创建一个 EMPLOYEE 表，其中包括一个 EMPLOYEE_ID 列，但不想每次向表中添加新雇员时都提供 EMPLOYEE_ID 值。这时可让 MS-SQL Server 在每次添加新行时提供“下一个”EMPLOYEE_ID 值，可使用类似于以下的 SQL 语句：

```
CREATE TABLE employee  
(id INTEGER IDENTITY(10,10),  
name VARCHAR(35),  
quota SMALLINT)
```

IDENTITY 属性的格式为：

```
IDENTITY (initial_value, increment)
```

如果忽略了 initial_value 和 increment 参数，则 MS-SQL Server 将把 initial_value 和 increment 值都设置为 1。

在本例的 CREATE TABLE 语句中, 告诉 MS-SQL Server 将 10 赋予添加到 EMPLOYEE 表中的第一行的 ID 列。然后, 当添加接下来的行时, MS-SQL Server 将向表中最新的一行的 ID 值上加 10 并将该值赋予添加进来的新行。这样一来, 执行以下 SQL 语句:

```
INSERT INTO employee (name, quota)
VALUES ('Sally Smith', NULL)
INSERT INTO employee (name, quota)
VALUES ('Wally Wells', 5)
INSERT INTO employee (name, quota)
VALUES ('Greg Jones', 7)
SELECT * FROM employee
```

MS-SQL Server 将在显示中插入 3 个雇员行并将其显示为类似于下面的样子:

id	name	quota
10	Sally Smith	NULL
20	Wally Wells	5
30	Greg Jones	7

可以只将 IDENTITY 属性运用于类型为 INTEGER、INT、SMALLINT、TINYINT、DECIMAL 或 NUMERIC 的列——而且仅当该列不允许 NULL 值时。

注意: 为一列指定 IDENTITY 属性并不能保证每行在该列有惟一的值。例如, 假设对本例中的表执行以下 SQL 语句:

```
SET IDENTITY_INSERT employee ON
INSERT INTO employee (id, name, quota)
VALUES(20, 'Bruce Williams', NULL)
SET IDENTITY_INSERT employee OFF
INSERT INTO employee (name, quota)
VALUES('Paul Harvey', 9)
SELECT * FROM employee
```

MS-SQL Server 将显示类似于下面的表行:

id	name	quota
10	Sally Smith	(null)
20	Wally Wells	5
30	Greg Jones	7
20	Bruce Williams	(null)
40	Paul Harvey	9

由于第一个 INSERT 语句指定了 ID 列的值, DBMS 就在 Bruce Williams 行的 ID 列中放入值 20。第二个 INSERT 语句并未在 ID 列包括一个值。结果, DBMS 向最高的 ID (30) 中加上 10 (增量) 并使用结果 40 作为新的 Paul Harvey 行的 ID 值。

如果想要保证 IDENTITY 列在表的每一行中包括惟一的值, 必须创建一个惟一的基于 IDENTITY 列的索引。

技巧 18 理解断言 (Assertions)

正如读者在技巧 11 “理解约束”中所学到的知识, 一个约束是限制用户或应用程序输入

到表中的列内数据的数据库对象。一个断言是用来检查约束以限制可向数据库中作为整体输入的数据值的数据库对象。

不管是断言还是约束都是作为检查条件而指定的，其条件可求值为 TRUE 或 FALSE 值。虽然约束使用检查条件对单个表起作用，限制赋予表中的列的值；断言中的检查条件涉及多个表和这些表中的数据关系。由于断言作为整体应用于数据库，因而可使用 CREATE ASSERTION 语句作为数据库定义的一部分而创建断言（作为对照，由于约束只应用于单个表，在创建表时可应用[定义的]约束）。

例如，如果想要防止投资者从保值基金中提取多于某一数量的款项，可使用以下 SQL 语句创建一个断言：

```
CREATE ASSERTION maximum_withdrawal
CHECK (investor.withdrawal_limit >
      SELECT SUM(withdrawals.amount)
      FROM withdrawals
      WHERE withdrawals.investor_id = investor.ID)
```

这样一来，创建断言的句法是：

```
CREATE ASSERTION <assertion name> <check condition>
```

当向数据库定义中添加了 MAXIMUM_WITHDRAWAL 断言时，DBMS 将检查以确保每次执行一个修改 INVESTOR 或 WITHDRAWALS 表的 SQL 语句时断言仍然是 TRUE。因而，每次用户或应用程序试图对在断言中有 CHECK 子句的表执行 INSERT、UPDATE、DELETE 语句时，DBMS 对数据库检查该检查条件，其中包括所提出的修改。如果检查条件仍然为 TRUE，则 DBMS 执行修改。如果修改使检查条件为 FALSE，则 DBMS 不能执行修改并返回一个错误代码，用以指出语句由于违反断言而不能成功执行。

技巧 19 理解 SQL DBMS 的客户/服务器模型

客户机/服务器计算（当使用 Internet 将客户连接到服务器时，常称为 n 层计算）涉及分布式数据处理或多台计算机一起工作来执行一系列的操作。在客户机/服务器模型中，客户（工作站）和服务器（DBMS）一起工作来执行创建对象并执行操作数据库中数据的操作。虽然以整体方案在一起工作，但服务器执行的任务与客户完成的工作是不同的。

关系 DBMS 模型和 SQL 都是特别适合于使用客户机/服务器环境的。DBMS 和驻留在中心服务器（计算机）上的数据以及多个客户（网络工作站）通过在局域网（LAN）上的连接向服务器传达对数据的请求。运行在客户机器上的应用程序接受用户输入并形成 SQL 语句，然后将其发送到服务器上的 DBMS。DBMS 再翻译并执行 SQL 命令，并将结果发送回客户（工作站）。最后，运行在工作站上的应用程序格式化并向用户显示结果。

使用 SQL 的客户机/服务器关系与简单的数据库文件共享系统（工作站从文件服务器复制大量数据并在本地操作数据，然后将大量数据发送回文件服务器，从而在网络磁盘驱动器上保存起来）对于带宽使用是特别有效的。换一种说法，老式的不太有效的共享文件访问方法涉及向用户发送所需的整个“文件柜”及其文件夹。

在客户机/服务器模型中，服务器检查“文件柜”并向应用程序只发送所希望使用的文件夹。用户使用一种运行在网络工作站上的应用程序（客户）向 DBMS（服务器）发送对数据的请求（使用 SQL 语句），DBMS 和数据驻留于同一系统上，因而 DBMS 可执行 SQL 语句并

通过 LAN 向工作站只发送用户所需的数据。

DBMS（服务器）在从客户接收到请求（一个或多个 SQL 语句）之前没什么事可做。服务器负责为多个客户存储、操作并提取数据。因而，服务器硬件通常是多个高端处理器，以便同时处理数据请求和大量的快速存储，服务器是为快速数据访问和提取而优化的。

当处理 SQL 语句时，DBMS（服务器）解释命令并将其翻译为数据库操作。在执行了操作之后，服务器再将结果格式化并发送给客户。这样一来，服务器的任务相对直接：读取、解释并执行 SQL 语句。另外，服务器没有向用户呈现信息的职责，把该任务留给了客户。

SQL 的客户机/服务器系统的客户部分由硬件（在处理能力上常常类似于服务器）和软件（即用户与 DBMS 之间的界面）组成。当处理 SQL 时，用户常常认识不到还有一台独立的 DBMS 服务器存在。说到用户，运行在他或她的计算机上的应用程序（如定单输入系统）是对存储在共享的网络驱动器上的数据操作的。事实上，客户（应用程序）接受用户输入、翻译用户输入到 SQL 命令中的信息并将命令与输入的数据一起发送到 DBMS 服务器。然后应用程序等待服务器送回结果，由程序再将结果显示给用户。

在客户机/服务器环境中，客户负责以下工作：

- 从用户（或另一应用程序）接受所需的信息
- 形成对服务器的数据提取、删除或是更新的请求
- 向用户显示所有信息（数据和服务器消息）
- 操作单独的数据项（服务器管数据的物理存储、删除和提取，但数据值却是在客户机/服务器模型的客户端确定的）
- 格式化信息并产生报告（既可是打印的也可是在线的）

注意：可通过在客户应用程序中重复数据确认检查来减少网络流量和服务器负载。例如，在向 DBMS 发送定单行中的列之前，让应用程序强制用户输入一个合法的量，从而避免向服务器发送数据，让 DBMS 分析 SQL 语句，只将无效的语句发送回客户端。

除了（但不代替）服务器的 SQL 定义的数据完整性机制之外，一定要在客户机/服务器模型的客户端使用确认检查。通过使用同样的一套规则在服务器上的确认（如果合适的话，也在客户上重复进行），从而保证“每个”应用程序的数据是有效的。如果信任应用程序执行其自己的确认，则必定将陷入某种问题，即由于在测试阶段忽略了确认代码不经意间跑到了工作系统之外。另外，如果必须改变或添加新的商业规则，在一处（在服务器上）改变服务器的确认检查，与和每个软件开发商（或是本单位的编程人员）联系以便更新单独的应用程序比起来也是相对简单的。

技巧 20 理解 SQL 语句的结构

在使用 SQL 语句向 DBMS 发送命令时，首先要告诉 DBMS 想要做什么，然后描述想要 DBMS 对其采取行动的数据（或结构）。图 1.9 显示的是 SQL 语句的基本形式。

如图 1.9 所示，每条 SQL 语句以描述语句要做什么的关键字开始。在 SQL 语句中找到的关键字包括：SELECT、INSERT、UPDATE、DELETE、CREATE 或 DROP。在告诉 DBMS 想要做什么之后，再告诉 DBMS 感兴趣的列和想要在其中查找的列的表。通常通过在动词之后列出想要使用的列（在 SQL 语句的开始），并在关键字 FROM 之后列出表。

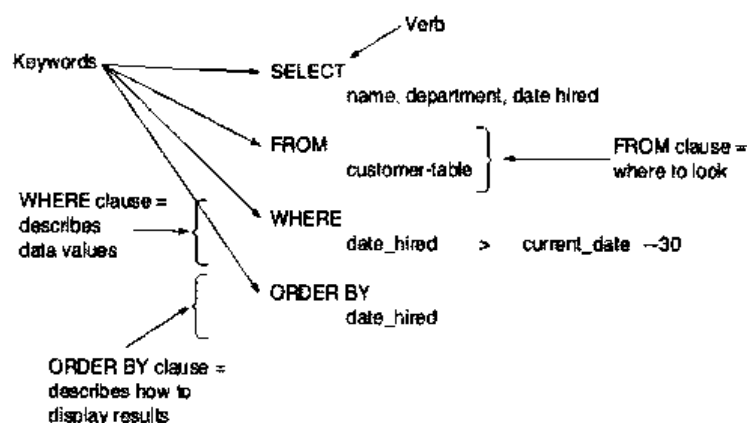


图 1.9 SQL 语句的基本结构

在告诉 DBMS 做什么以及确定了要对其操作的列和表之后，可用一个或多个子句结束 SQL 语句，这些子句既可进一步描述 DBMS 所采取的行动，也可对确定想要 DBMS 对其采取行动的表行的数据值给出描述。通常的描述子句以关键字 HAVING、IN、INTO、LIKE、ORDER BY、WHENEVER、WHERE 或 WITH 开始。

ANSI/ISO SQL-92 大约有 300 个保留字，其中大约有 30 个可用来对数据库做其主要的工作。表 1.1 列出了某些最常用的关键字的清单。虽然某些关键字只适用于 MS-SQL Server，但如果使用另一种 DBMS 产品的话，也会找到执行类似功能的关键字。

表 1.1 最常用的 SQL 和 MS-SQL Server 关键字

关键字	描述
数据定义语言 (DML)	
CREATE DATABASE	(MS-SQL Server)。创建一个数据库及其事务处理日志。数据库有一个或多个架构，其中包括诸如表、视图、域、约束、例程、触发器等数据库对象
DROP DATABASE	(MS-SQL Server)。删除数据库及其事务处理日志
CREATE SCHEMA	向数据库中添加一个命名的数据库对象容器。可以有不止一个架构。所有数据库对象（表、视图、域、约束、例程、触发器等）驻留于数据库的架构之一中
DROP SCHEMA	从数据库中删除数据库的架构
CREATE DOMAIN	为数据库表中的列创建允许值的清单。可将域用作多个表中的列的数据类型
DROP DOMAIN	从数据库中删除域定义
CREATE TABLE	创建保存数据的列和行的结构（表）
ALTER TABLE	向表中添加列，从表中删除列，改变列的数据类型或者向表中添加列约束
DROP TABLE	从数据库中删除表
CREATE VIEW	从一个或多个表中创建显示一系列或多列的行。某些视图允许更新基表
DROP VIEW	丢弃数据库视图
CREATE INDEX	创建一个结构，其中有来自表列的值，这可加速 DBMS 在表内找到特定行的能力
DROP INDEX	从数据库中删除 INDEX（索引）

续表

关键字	描述
数据操作语言 (DML)	
INSERT	向表中添加一个或多个行
SELECT	提取数据库中的数据
UPDATE	更新表中的数据值
DELETE	从表中删除一行或多行
TRUNCATE	(MS-SQL Server)。从表中删除所有行
数据控制语言 (DCL)	
ROLLBACK	取消直到最近的 COMMIT 或 SAVEPOINT 之后对数据库对象所做的改变
COMMIT	使对数据为所做的提议中的改变成为永久的 (提交了改变不能用 ROLLBACK 命令取消)
SAVEPOINT	在事务处理 (即一系列操作) 中标记一个点, 可用于 ROLLBACK 命令中, 以便取消部分事务处理, 而不取消整个事务处理
GRANT	授予对数据库对象或 SQL 语句以访问权
REVOKE	删除对数据库对象或执行特定的 SQL 语句的访问权限
编程的 SQL	
DECLARE	保留服务器资源以备临时表 (cursor) 所用
OPEN	创建一个临时表并用一个或多个数据库表从一行或多行中选择的数据值加以填充
FETCH	从临时表中向主变量传递数据值
CLOSE	释放用于保存从数据库复制到临时表中的数据所使用的资源
DEALLOCATE	释放保留为临时表所用的服务器资源
CREATE PROCEDURE	(MS-SQL Server)。创建 SQL 语句的命名的列表, 具有适当访问权限的用户可使用其名称来执行其中的语句, 就好像用户有另外的 SQL 关键字一样
ALTER PROCEDURE	(MS-SQL Server)。当用户调用一个例程时, 改变 DBMS 将执行的 SQL 语句的顺序
DROP PROCEDURE	(MS-SQL Server)。从数据库中删除一个例程
CREATE TRIGGER	(MS-SQL Server、DB2、PL/SQL)。创建命名的 SQL 语句的序列, 当一系列具有特定的数据值或当用户试图执行一个特定的数据库命令 (触发的事件) 时, DBMS 将自动执行此语句序列
ALTER TRIGGER	(MS-SQL Server、DB2、PL/SQL)。当 DBMS 删除触发的事件或改变事件的性质时, 改变所执行的 SQL 语句
DROP TRIGGER	(MS-SQL Server、DB2、PL/SQL)。从数据库中删除触发器
DESCRIBE INPUT	在动态的 SQL 语句执行期间, 保留应用程序将用来向 DBMS 传递数据的输入区域
GET DESCRIPTOR	指示 DBMS 使用 DESCRIPTOR 区域来提取由应用程序在动态的 SQL 语句执行期间放置在该处的数据值
DESCRIBE OUTPUT	在动态的 SQL 语句执行期间, 保留 DBMS 将用来从数据库向应用程序传递数据的输出区域

续表

关键字	描述
编程的 SQL	
SET DESCRIPTOR	指示 DBMS 将数据放入 DESCRIPTOR 区域, 以便在动态的 SQL 语句执行期间由应用程序加以提取
PREPARE	指示 DBMS 在动态的 SQL 语句执行期间创建一个执行计划或编译 SQL 语句
EXECUTE	指示 DBMS 执行一条动态的 SQL 语句

技巧 21 使用 MS-SQL Server Query Analyzer 执行 SQL 语句

可使用 MS-SQL Server Query Analyzer (QA) 来执行 MS-SQL Serve 支持的任何 SQL 语句 (正如前面技巧中提过的, 没有一个商业数据库支持 SQL-92 标准的所有内容)。QA 具有图形用户界面(GUI), 可用来提出实际(交互式)的查询并向 MS-SQL Server 发送 SQL 命令(MS-SQL Server 还通过 ISQL 提供与数据库连接的命令行界面, 读者将在技巧 22 “使用 MS-SQL Server ISQL 在命令行上执行 SQL 语句或是存储在 ASCII 文件中的语句”中学习有关 ISQL 的知识)。

注意: 在使用 Query Analyzer 之前必须安装 MS-SQL Server。

为了启动 MS-SQL Server QA, 可执行以下步骤:

1. 单击 Start 按钮, Windows 将显示 Start 菜单。
2. 选择 Programs→Microsoft SQL Server 7.0, 单击 Query Analyzer。Windows 将启动 QA 并显示 Connect to SQL Server (与 SQL Server 连接) 对话框, 如图 1.10 所示。

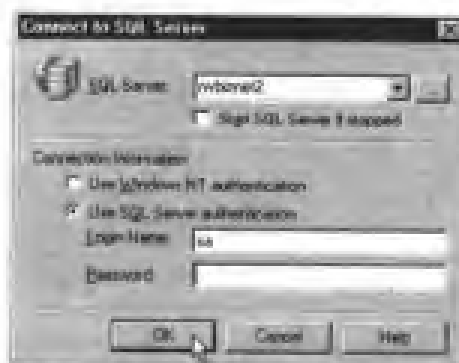


图 1.10 MS-SQL Server Query Analyzer 的 Connect to SQL Server 对话框

3. 在 SQL Server 字段中键入想连接其上的 SQL Server 名称 (SQL Server 名称通常与在其上安装 MS-SQL Server 的 Windows NT Server 名称相同)。

4. 在 Login Name (登录名) 字段中键入登录名。当安装 MS-SQL Server 时, 程序自动创建一个没有密码的 sa (系统管理员) 账户。如果正在自己安装的 MS-SQL Server 上工作的话, 可使用 sa 账户; 如果不是, 键入 Login Name 和系统管理员 (或数据库管理员) 给予的密码。

5. 单击 OK 按钮。QA 将登录到步骤 4 中指定的 MS-SQL Server 上并在 QA 应用程序窗口中显示 Query (查询) 窗格, 该窗格与图 1.11 所示的类似。

当在 Windows NT 下安装 MS-SQL Server 时, 安装程序创建几个数据库, 如图 1.11 中 Query 窗格的右边角中的 DB 下拉列表所示。在使用 QA 向 MS-SQL Server 发送 SQL 语句之前, 必须选择一个数据库。

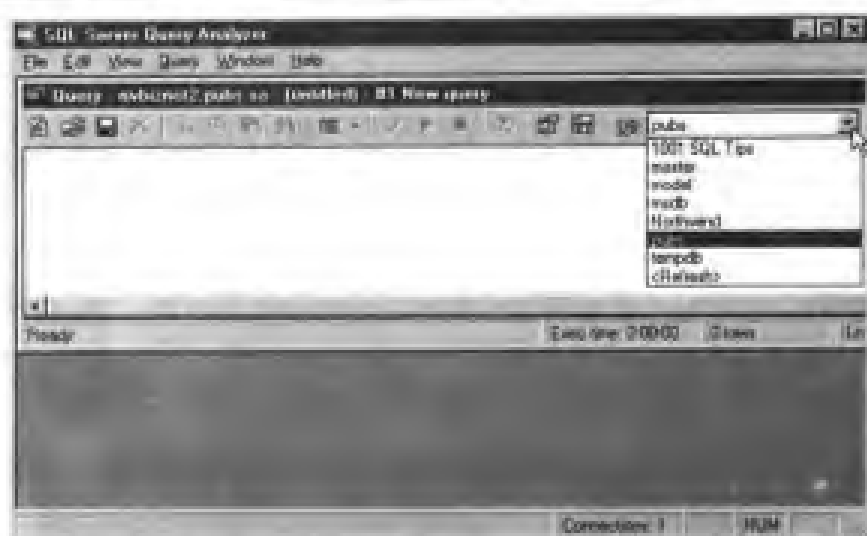


图 1.11 MS-SQL Server Query Analyzer 在 Query Analyzer 窗口中的 Query 窗格

为了处理 pubs（示例）数据库，可执行以下步骤：

1. 单击 DB 字段右边的下拉按钮（在 QA 的 Query 窗格右上角），列出连接其上的 SQL Server 上的数据库。
2. 单击一个数据库将其选择。对于本例来说，单击 pubs。
3. 单击 Query 窗格中的任意一处将光标放在 Query 窗格中。QA 将把光标放在 Query 窗格的左上角。
4. 在 Query 窗格中键入 SQL 语句。对本例来说，键入 `SELECT * FROM authors`。
5. 为了执行查询（在第 4 步中键入的），既可按 F5 键，也可按 Ctrl+E，或选择 Query 菜单中的 Execute 选项。对本例来说，按 F5 键。QA 将在 Query 窗格之下的 Results（结果）窗格中显示查询结果，如图 1.12 所示。

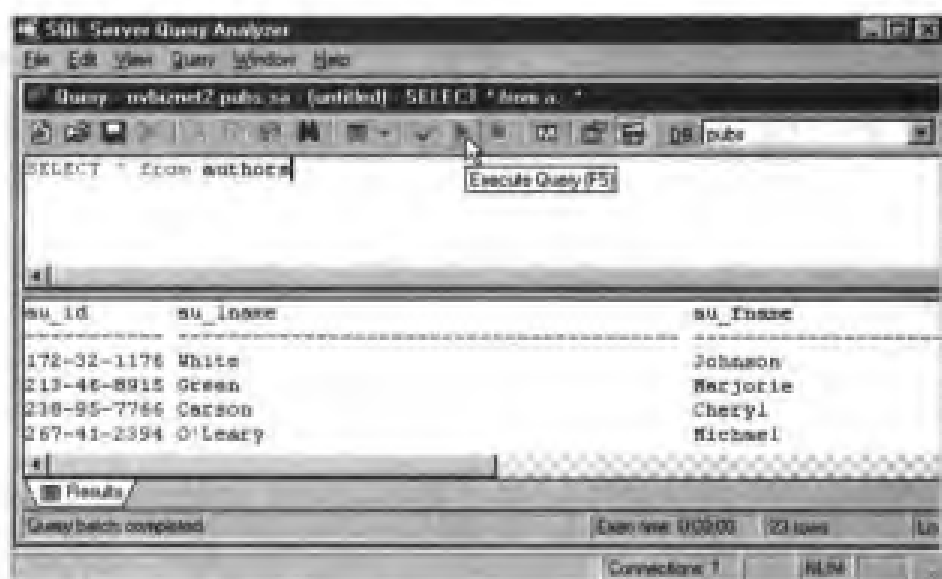


图 1.12 Query Analyzer，查询在 Query 窗格，而查询结果在 Results 窗格

无论何时指示 QA 来执行 Query 窗格中的 SQL，除非选择了想要执行的特定语句（或一组语句），否则 QA 将向 SQL Server 发送 Query 窗格中的所有语句以便处理。因而，一定要小

心，不要认为 QA 只向 SQL Server 发送键入的最后一条语句而按 Ctrl+E（或按 F5 键，或选择 Query 菜单的 Execute 选项）。

如果在 Query 窗格中有多条语句，既可删除那些不想执行的语句，也可高亮显示想要 QA 向 SQL Server 发送用以处理的语句。例如，如果在 Query 窗格中有以下语句：

```
SELECT * FROM authors
```

```
SELECT * FROM authors WHERE au_lname = 'Green'
```

而只想要执行第二条语句，则可高亮显示第二条查询将其选择，然后选择 Query 菜单的 Execute 选项（或在标准工具栏上单击绿色的 Execute Query[执行查询]按钮）。QA 将只向 SQL Server 发送所选的第二条语句并在 Results 窗格中显示结果，如图 1.13 所示。

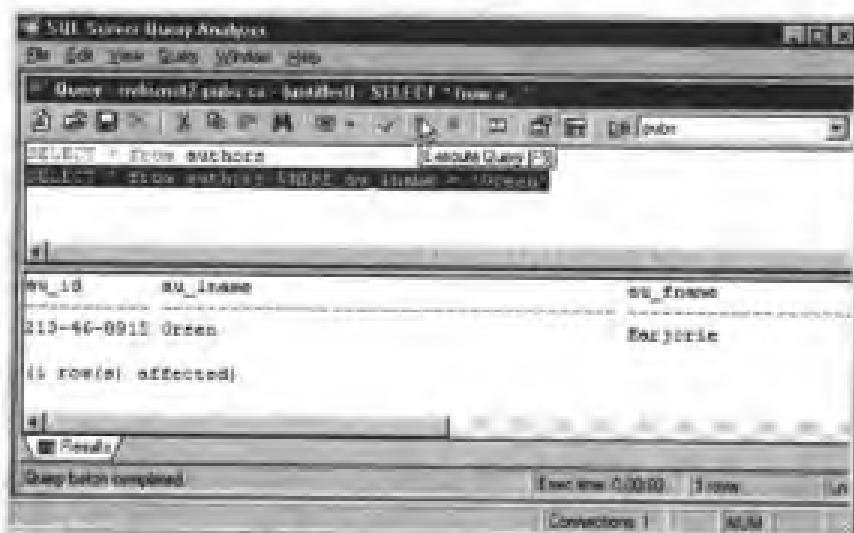


图 1.13 当 Query 窗格中包括多条语句时，在执行了所选的语句之后的 Query Analyzer

在执行了语句之后使 QA 在 Query 窗格中保留 SQL 语句，可节约大量的键入工作，特别是，如果键入复杂的查询且未获得所期望的结果时更是如此。如果需要改变查询的选择子句中的逻辑，只需在 SQL 语句中单击光标并作出改变，而不必重新键入整个语句。

技巧 22 使用 MS-SQL Server ISQL 在命令行上执行 SQL 语句或是执行存储在 ASCII 文件中的语句

在技巧 21 “使用 MS-SQL Server Query Analyzer 执行 SQL 语句”中，读者学习了如何使用 MS-SQL Query Analyzer 的 GUI 查询工具。MS-SQL Server 还包括两个命令行查询工具：ISQL.EXE 和 OSQL.EXE。可以在 MS-SQL Server 的 BINN 子目录中找到这两个工具（如果将 MS-SQL Server 安装到默认的 C:\MSSQL7 文件夹中，将在 C:\MSSQL7\BINN 子文件夹中找到 ISQL 和 OSQL）。

在 ISQL 和 OSQL 之间，除了名称之外，其不同之处在于 ISQL 使用 DB-LIB 与数据库连接，而 OSQL 使用 ODBC 与数据库连接。虽然我们将在本技巧中使用 ISQL 来访问数据库，要了解的重要事情是，使用 OSQL 可执行同样的语句。因而，如果在自己的系统上只有 OSQL，在以下的例子中只要用它来代替 ISQL 即可。

如果要运行一系列的 SQL 语句，命令行查询工具是有用的。可用 ISQL（或 OSQL）来执行一条接一条地执行语句，只要将其键入将传递到 ISQL 或 OSQL 用于处理的 ASCII 文件中即可。

完成步骤 4 之后，ISQL 将把在 GO 命令之前键入的 SQL 语句发送到 DBMS，显示结果，然后显示另一准备好提示（1>）指出它已准备好接受下一条命令。

要理解的重要事情是，ISQL 只在准备好提示下键入 GO 并按 Enter 键之后才向 SQL Server 发送 SQL 语句。

为了退出 ISQL，可在准备好提示下键入 EXIT，然后按 Enter 键。ISQL 将终止，计算机将返回到 MS-DOS 提示下。

为了退出 MS-DOS 会话并返回 Windows 桌面，可在 MS-DOS 提示下键入 EXIT 并按 Enter 键。

注意：在步骤 2 中键入 USE pubs，就告诉 DBMS 想要使用 PUBS 数据库。不用让 ISQL 向 DBMS 发送 USE 语句，在启动 ISQL 时可添加 -d<use database name> 来选择想要使用的数据库。在本例中可键入：

```
ISQL -SNVBizNet2 -Usa -P -dpubs
```

为了启动 ISQL，可登录到名为 NVBizNet2 的 SQL Server 的 sa 账户上并选择 PUBS 作为下一 SQL 语句要使用的数据库。

正如在本技巧的开始所提到的，可将 SQL 语句键入 ASCII 文件，然后让 ISQL（或 OSQL）执行这些语句。为达到此目的，在键入 ISQL 的启动命令时，应添加 -i<input file> 参数。例如，假设在名为 INFILE39.SQL 的文件中有以下语句：

```
USE pubs
SELECT au_ID, au_lname, zip FROM authors WHERE zip = 94301
GO
```

通过用以下命令行启动 ISQL，可告诉 ISQL 向 DBMS 发送 INFILE39.SQL 中的两条语句并在屏幕上显示结果：

```
ISQL -SNVBizNet2 -Usa -P -dpubs -iInFile39.sql -n
```

-n 告诉 ISQL 不要显示语句号。没有 -n，ISQL 将对三条 SQL 语句中的每一条显示一个语句号和大于号（>）。结果，标题没有与列数据列在一起。-n 告诉 ISQL 不要显示语句行号。在键入命令行之后，按 Enter 键。ISQL 将把输入文件 InFile39.sql 中的每条语句发送到 DBMS 并显示如下的输出：

au_ID	au_lname	zip
427-17-2319	Dull	94301
846-92-7186	Hunter	94301

作为最后的变化，为了将查询结果保存到一个文件中而不是显示在屏幕上，可在 ISQL 的启动命令之后添加 -o<output file> 参数。例如，假设想要将查询结果保存到输出文件 OUTFLE39 中。可在 MS-DOS 提示符下键入以下命令，然后按 Enter 启动 ISQL。

```
ISQL -SNVBizNet2 -Usa -P -iInFile39.sql -n -oOutFile39
```

技巧 23 在 ISQL 内使用 ED 命令编辑 SQL 语句

当键入 GO 命令将 SQL 语句发送给 DBMS 之前，ISQL 就好像是一个行编辑程序。正如在技巧 22 “使用 MS-SQL Server ISQL 在命令行上执行 SQL 语句或是执行存储在 ASCII 文件中的语句” 中所学到的，在 MS-DOS 命令行上启动 ISQL 的命令格式是：

```
ISQL -S<server name> -U<username> -P<password>
```

注意：如果 sa 账户不可用，可用实际的 SQL Server 名代替 NVBizNet2，并使用自己的用户名和密码登录。

这样一来，为了以用户名 sa 登录到名为 NVBizNet2 的 MS-SQL Server 上，可执行以下步骤：

1. 单击 Start 按钮，选择 Programs，然后单击 Command Prompt（命令提示），Windows 将启动一个 MS-DOS 会话。

2. 为了启动 ISQL，键入 ISQL -SNVBizNet2 -Usa -P，然后按 Enter 键。ISQL 将显示其准备好提示（1>）。

3. 接着，键入 SQL SELECT 语句：

```
SELECT * FROM authors
```

4. 按 Enter 键，在 ISQL 将语句放入语句缓冲区之后，屏幕看起来与下面类似：

```
ISQL -SNVBizNet2 -Usa -P
```

```
1> SELECT * FROM authors
```

```
2>
```

由于没有确定要使用的数据库，如果在准备好提示（2>）后键入 GO 并按 Enter 键，则 ISQL 将显示以下内容：

```
Msg 208, Level 16, State 1, Server NVBIZNET2, Line 1
```

```
Invalid object name 'authors'.
```

由于 ISQL 是个行编辑器界面，不能将光标移动到 SELECT 之前并插入语句。因而，如果只有 ISQL，那么惟一的选择是在准备好提示（2>）后键入 EXIT 或 QUIT 并再次启动，这一次既可向 ISQL 命令行添加 -d <use database name> 参数，也可在响应第一个准备好提示（1>）时键入 USE <database name>。

幸运的是，ISQL 允许使用 MS-DOS 的全屏幕编辑器。

为了启动全屏幕编辑器，在准备好提示（2>）下键入 ED 命令并按 Enter 键。ISQL 将启动 MS-DOS 编辑器并将语句缓冲区中的内容转移过来，如图 1.15 所示。



图 1.15 由 ISQL 启动的 MS-DOS 全屏幕编辑器

为了在 SELECT 语句之前插入 USE 语句，可将光标移到 SELECT 之前。键入 USE pubs 并按 Enter 键。当做了这些之后，在文本编辑器中将会有两条语句：

```
USE pubs
```

```
SELECT * FROM authors
```

为了将全屏幕编辑器的内容转移到 ISQL 的语句缓冲区中，可选择 File、Exit 选项。当编辑器提示保存改变时，按 Y 键。MS-DOS 编辑器将把其内容发送到 ISQL，ISQL 将把内容显示为单独的行，类似于：

```
1> USE pubs
2> SELECT * FROM authors
3>
```

现在键入 GO 并按 Enter 键，将 USE 和 SELECT 语句发送到 DBMS。在 ISQL 显示查询结果之后，键入 EXIT 并按 Enter 键退出 ISQL 返回 MS-DOS 提示。

现在要了解的重要事情有：

- 可在单行编辑模式下键入 SQL 语句以响应每个 ISQL 的准备好的提示符。
- ISQL 将所键入的每条语句保存在语句缓冲区中。
- 在响应 ISQL 的准备就绪提示时键入 ED 可使用全屏幕编辑器。
- 启动了全屏幕编辑器（用 ED 命令）之后，ISQL 将其语句缓冲区的内容复制到编辑器屏幕上。
- 当离开全屏幕编辑器（选择 File 菜单上的 Exit 选项）时，ISQL 将编辑器屏幕内容读到语句缓冲区中，每个编辑器行作为一条语句。

技巧 24 使用 CREATE DATABASE 语句创建 MS-SQL Server 数据库和事务处理日志

与许多其他 DBMS 产品不同，MS-SQL Server 允许为每个 MS-SQL Server 创建多个数据库。大多数商业 DBMS 产品甚至没有 CREATE DATABASE 命令。相反，安装程序创建 SQL Server 将要使用的数据库文件。数据库管理员（dba）和有权限的用户再在一个数据库内创建所有的数据库对象。结果，通常数据库既包括有关系的也包括无关系的表的混合。

MS-SQL Server 给出了最好的结果。如果愿意的话，可为所有的表创建单个的数据库，也可以将完全无关的表分离到另一独立的数据库中。例如，假设用户及其老板开办自己的家庭商行。使用通常的 DBMS，就要创建一个数据库，其中既保存着用户自己（邮件定单）的 CUSTOMER 清单，也保存着老板（会计）的 CLIENT 清单，即使两个表是完全无关的。

具有单个数据库意味着，在备份和（如果需要的话）恢复操作时，两个商行都要失去数据库访问手段。如果两者是分开的，仍然可使用单一服务器（为了在软件和硬件上节约一点难挣的钱），但不会受与自己的表毫无关系的数据库问题或维护行为的影响。

最后，MS-SQL Server 的多个数据库策略使创建一个开发数据库成为可能，这个开发数据库与其工作副本使用同样的数据库对象和安全性设置。具有一样的数据库结构和安全性设置更易于测试所建议的改变如何影响在线的应用程序、数据库的存储过程、视图和触发器。另外，在开发系统上充分地测试新的或修改了的代码之后，将能够在工作副本上安装过程、触发器和视图，而不用进一步的修改。最后，可以从工作数据库中将表中的数据导入到开发数据库的同样的表中，使之易于使用开发系统来“冻结”数据库数据并再现那些似乎是随机出现的错误。

CREATE DATABASE 语句的句法如下：

```
CREATE DATABASE <database name>
[ON { [PRIMARY] <filespec> } [, ... <last filespec> }]
[LOG ON { <filespec> } [, ... <last filespec> ]]
```

[FOR RESTORE]

其中<filespec>定义为:

```
(NAME = <logical file name>,  
FILENAME = '<physical file name>'  
[, SIZE = <initial file size>]  
[, MAXSIZE = (<maximum file size> | UNLIMITED)]  
[, FILEGROWTH = <file extension inc>])
```

请审视一下表 1.2, 从中可获得 CREATE DATABASE 及各关键字和选项的解释。

表 1.2 CREATE DATABASE 语句中关键字和选项的定义

关键字/选项	描述
Database name	数据库的名称
ON <filespec>	将要保存数据库的数据部分的磁盘文件名。MS-SQL Server 允许用户将单个数据库分成多个文件
PRIMARY	如果将数据库分成多个文件, PRIMARY 确定包括数据开始和系统表的文件。如果不指定一个 PRIMARY 文件, MS-SQL Server 将使用列表中的第一个文件作为 PRIMARY 文件
LOG ON <filespec>	将保存事务处理日志的磁盘文件名
FOR RESTORE	在使用由 RESTORE 操作的数据填充数据库之后才允许访问数据库
<logical file name>	MS-SQL Server 将用来引用数据库或事务处理日志的名称
<physical file name>	指向数据库或事务处理日志的全路径
<initial file size>	以 MB 量度的数据库或事务处理日志的起始尺寸。如果对事务处理日志未指定起始尺寸, 系统将把其尺寸设置为总数据库文件尺寸的 25%
<maximum file size>	数据库和事务处理日志可增长到的最大尺寸。如果指定为 UNLIMITED, 文件可增长到消耗全部的物理磁盘空间的程度
<file extension inc>	当前文件中的自由空间用光时, 可增加到事务处理日志或数据库文件中的字节数

为了使用 MS-SQL Server Query Analyzer 来创建数据库, 可执行以下步骤:

1. 单击 Start 按钮, Windows 将显示 Start 菜单。
2. 将鼠标指针移动到 Start 菜单的 Programs 上, 选择 Microsoft SQL Server 7.0 选项, 单击 Query Analyzer。Query Analyzer 将显示 Connect to SQL Server (与 SQL Server 连接) 对话框, 如图 1.16 所示。

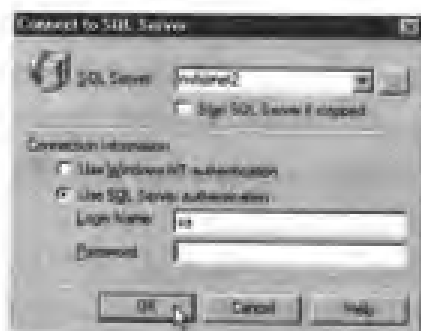


图 1.16 Query Analyzer 的 Connect to SQL Server 对话框

3. 在 SQL Server 字段中键入自己的 SQL Server 的名称。
4. 在 Login Name 字段键入用户名并在 Password 字段内键入密码。
5. 单击 OK 按钮。Query Analyzer 将与步骤 3 中键入的 SQL Server 连接并在 SQL Server 的 Query Analyzer 应用程序窗口中显示 Query 窗格。

6. 键入 CREATE DATABASE 语句。对于本例来说，键入：

```
CREATE DATABASE SQLTips
ON (NAME = SQLTips_data,
    FILENAME = 'c:\mssql7\data\SQLTips_data.mdf',
    SIZE = 10,
    FILEGROWTH = 1MB)
LOG ON (NAME = 'SQLTips_log',
    FILENAME = 'c:\mssql7\data\SQLTips_log.ldf',
    SIZE = 3,
    FILEGROWTH = 1MB)
```

7. 单击标准工具栏上绿色箭头的 Execute Query（执行查询）按钮（或选择 Query 菜单的 Execute 选项）。Query Analyzer 将在步骤 5 中连接上的 SQL Server 上创建数据库。

在完成了步骤 7 之后，Query Analyzer 将在 SQL Server Query Analyzer 应用程序窗口的 Results 窗格中显示 CREATE DATABASE 语句的执行结果。如果 Query Analyzer 成功地执行了 CREATE DATABASE 语句，该程序将在 Results 窗格中显示如下信息：

```
The CREATE DATABASE process is allocating 10.00 MB on disk 'SQLTips_data'.
The CREATE DATABASE process is allocating 3.00 MB on disk "SQLTips_log".
```

技巧 25 使用 MS-SQL Server Enterprise Manager 创建数据库和事务处理日志

在技巧 24 “使用 CREATE DATABASE 语句创建 MS-SQL Server 数据库和事务处理日志”中读者已经知道，MS-SQL Server 允许用户在单个服务器上创建多个数据库，而且还学习了如何使用 CREATE DATABASE 语句。与大多数数据库管理工具类似，MS-SQL Server 不仅给出了命令行（SQL 或 Transact-SQL）语句，而且还有图形用户界面（GUI）工具可以执行同样的功能。为了使用 MS-SQL Server Enterprise Manager 来创建数据库，可执行以下步骤：

1. 单击 Start 按钮，Windows 将显示 Start 菜单。
2. 将鼠标指针移动到 Start 菜单中的 Programs 上，选择 Microsoft SQL Server 7.0 选项并单击 Enterprise Manager。Windows 将在 SQL Server Enterprise Manager 应用程序窗口中启动 Enterprise Manager。
3. 单击 SQL Server Group 左边的加号（+）显示在网络上可用的 MS-SQL Servers 的清单。
4. 单击想要在其上创建数据库的 SQL Server 左边的加号（+）。接着 Enterprise Manager 将显示如图 42.2 所示的 Database Properties（数据库属性）对话框。
5. 单击 Databases 文件夹显示在当前服务器上的数据库的列表，如图 1.17 所示。
6. 选择 Action（操作）菜单中的 New Database（新数据库）选项。Enterprise Manager 显示 Database Properties 对话框，如图 1.18 所示。
7. 在 Name 字段中键入数据库名。对于本例来说，键入 MARKETING。Enterprise Manager 将自动地填充 Database Properties 对话框的 Database Files（数据库文件）部分的路径和初始数据库尺寸。

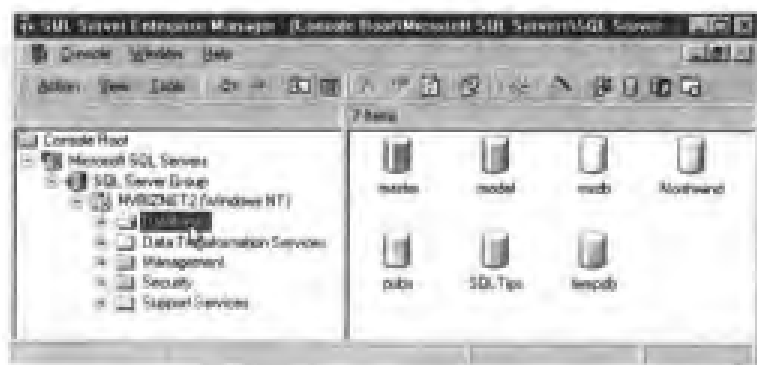


图 1.17 SQL Server Enterprise Manager 应用程序窗口



图 1.18 Enterprise Manager 的 Database Properties 对话框

注意：如果想要把数据库放入与默认的文件夹不同的文件夹中或是改变物理文件名，可单击 Database Properties 对话框的 Database Files 区域中的 Location(位置)字段中的 Search(搜索)按钮。Enterprise Manager 将显示 Locate Database File(查找数据库文件)对话框，这样就可选择一个文件夹或改变数据库的物理文件名。

8. 单击 Initial size (MB) 字段并键入数据库文件的起初尺寸。对本例来说，可键入 10。

9. 设置数据库 File Growth(文件增长)和 Maximum File Size(最大文件尺寸)选项。对于本例来说，可接受其默认值，即允许数据库文件每次填充增长 10%，而对最大尺寸没有限制。

10. 单击 Transaction Log(事务处理日志)选项卡。

11. 如果想要改变事务处理日志文件的路径(文件名和物理位置)，可单击 Location 字段中的 Search 按钮并处理 Locate Transaction Log File(查找事务处理日志文件)对话框。对于本例来说，接受事务处理日志的默认路径。

12. 单击 Initial size (MB) 字段，键入事务处理日志的初始尺寸。对于本例来说，键入 3。

13. 设置数据库的 File Growth 和 Maximum File Size 选项。对于本例来说，接受其默认值，即允许数据库文件每次填充增长 10%，而对最大尺寸没有限制。

14. 单击 OK 按钮。

在完成了步骤 14 之后，Enterprise Manager 将根据所选的选项创建数据库(在本例中是

MARKETING) 并返回 SQL Server Enterprise Manager 应用程序窗口。

现在要了解的重要事情是, MS-SQL Server 为用户提供了两种创建数据库的方式。可使用 CREATE DATABASE 语句或使用 Enterprise Manager 的 Action 菜单中的 New Database 选项。不管是使用 CREATE DATABASE 还是 Enterprise Manager, 都可设置指定以下事项的数据库和事务处理日志的选项:

- 数据库和事务处理日志文件的物理位置 (路径)
- 数据库和事务处理日志的初始尺寸
- 数据库和事务处理日志增长时的增量
- 数据库和事务处理日志增长的最大尺寸

如果正在使用 CREATE DATABASE 语句, 可在语句中独立的子句中指定数据库和事务处理日志的属性。当使用 Enterprise Manager 来创建数据库时, 可使用指定数据库选项的 Database Properties 对话框的 General 选项卡及使用 Transaction Log 选项卡来选择事务处理日志选项, 从而为数据库和事务处理日志文件指定不同的属性。

技巧 26 使用 DROP DATABASE 删除 MS-SQL Server 数据库和事务处理日志

删除不再需要的数据库可腾出磁盘空间。这时要遵循的主要原则是: 一定要小心! 因为不能容易地取消已经执行的 DROP DATABASE 语句。由此, 在删除数据库之前一定要将其备份。如果用户决定他或她又需要从数据库中得到某种信息 (当然是在刚刚删除时), 有了完整的备份则将省掉许多头痛的事情。

只有系统管理员 (sa) 或是具有 dbcreator 或 sysadmin 权限的用户才可删除数据库, 但不能删除 MASTER、MODEL 或 TempDB 数据库。

DROP DATABASE 语句的句法如下:

```
DROP DATABASE <database name>  
[,<database name>,<last database name>]
```

因而, 为了删除在技巧 25 “使用 MS-SQL Server Enterprise Manager 创建数据库和事务处理日志” 中创建的数据库 MARKETING, 可执行以下步骤:

1. 启动 MS-SQL Server Query Analyzer (正如在技巧 21 “使用 MS-SQL Server Query Analyzer 执行 SQL 语句” 中所学的), 或是启动 Enterprise Manager 并选择 Tools (工具) 菜单中的 SQL Server Query Analyzer 选项。
2. 在 Query Analyzer 的 Query 窗格中键入 DROP DATABASE 语句。对于本例来说, 键入:
DROP DATABASE marketing.
3. 按 Ctrl+E (或选择 Query 菜单、Execute 选项)。

在完成步骤 3 之后, Query Analyzer 将试图删除数据库及其日志文件。如果 Query Analyzer 成功地删除了 MARKETING 数据库及其事务处理日志, 将在 Query Analyzer 应用程序窗口的 Results 窗格中显示如下信息:

```
Deleting database file 'C:\MSSQL7\data\MARKETING_Data.MDF'  
Deleting database file 'C:\MSSQL7\data\MARKETING_Log.LDF'
```

技巧 27 理解如何确定 MS-SQL Server 数据库及其事务处理日志的容量

MS-SQL Server 将所有数据库对象 (表、视图、例程、触发器、索引等) 都放在单个的大

文件中。只要对数据库做出改变（添加对象、修改对象、删除一行、更新列值、插入一行等），DBMS 都在第二个文件（事务处理日志文件）中写入一个条目。这样一来，每个数据库都有两个文件：包括所有数据库对象的数据库文件和包括记载着对数据库所做的每条改变（从上次日志创建时开始）的条目的事务处理日志文件。

注意：数据库文件和事务处理日志每个都可由不止一个物理文件组成。但是，DBMS 将这一系列用于保存数据库数据的物理文件看作是一个单个文件，即逻辑的“数据库文件”，把一系列保存着事务处理日志的物理文件看作是一个单个文件，即逻辑的“事务处理日志”文件。用户可设置每个单独文件的初始容量，FILEGROWTH 选项适用于逻辑的数据库文件和事务处理日志，而不适用于每个在磁盘上保存其内容的物理文件。

正如在技巧 24 “使用 CREATE DATABASE 创建 MS-SQL Server 数据库和事务处理日志”和技巧 25 “使用 MS-SQL Server Enterprise Manager 创建数据库和事务处理日志”中所学的，可在创建时使用 SIZE 选项指定数据库和事务处理日志的初始容量。例如，在技巧 24 中，曾经执行过以下 SQL 语句：

```
CREATE DATABASE SQLTips
ON  (NAME = SQLTips_data,
     FILENAME = 'c:\mssql7\data\SQLTips_data.mdf',
     SIZE = 10,
     FILEGROWTH = 1MB)
LOG ON (NAME = 'SQLTips_log',
        FILENAME = 'c:\mssql7\data\SQLTips_log.ldf',
        SIZE = 3,
        FILEGROWTH = 1MB)
```

这就创建了 SQLTips 数据库文件（SQLTIPS_DATA.MDF），其初始容量为 10MB，也创建了该数据库的事务处理日志（SQLTIPS_LOG.LDF），其初始容量为 3MB。当向数据库的表中添加行时，会用光数据库文件的自由空间。如果向表中添加数据，其中每一行由 10 列 CHAR(100) 类型组成，每当向表中添加一行时就用掉 10MB 的 1000 个字节（10 列×100 字节/列）。

当已经使用了数据库文件所有的自由空间（在本例中为 10MB）时，就不能再向数据库添加数据了，即使有大量可用的物理磁盘空间也无济于事。为了避免在耗尽物理磁盘空间之前用光数据库文件的空间，在创建数据库时可使用 FILEGROWTH 选项。该选项告诉 MS-SQL Server 每当充满之后扩展数据库文件的容量。

在本例中，可将 FILEGROWTH 选项设置为 1MB，这就意味着，每当用光分配给数据库文件的空间时，DBMS 将该文件的尺寸增加 1MB。另外，由于并未指定最大的数据库文件容量，DBMS 将每次（需要时）把数据库文件扩展 1MB，直到耗尽物理磁盘空间为止。

每当对数据库做出改变时，DBMS 都在事务处理日志中保存原始数据值并详细地记下都做了什么。因此，如果对数据库做了大量的改变，DBMS 可能会相当快地用光分配给本例中的事务处理日志的 3MB 空间。幸运的是，可让 MS-SQL Server 扩展事务处理日志的容量，正如对数据库文件所做的一样。

在本例中，每当事务处理日志充满时，DBMS 都向事务处理日志增加 1MB 的自由空间。

注意：虽然本例对数据库文件和事务处理日志使用了相同的 FILEGROWTH 值，但两者是独立的。例如，可将数据库文件的 FILEGROWTH 值设置为 5MB，而将事务处理日志的

FILEGROWTH 值设置为 3MB——一个并不依赖于另一个。

一定要指定足够大的初始数据库文件容量和增长因子，从而当添加表行时 DBMS 不会花大量的时间来扩展其数据库文件的容量。为了确定初始数据库文件容量，可按以下步骤执行对数据库中每个表的分析：

1. 列出列名、数据类型和 DBMS 为了在列中保存一个值所需的磁盘空间字节数（系统手册将有 DBMS 所支持的数据类型所需的存储空间的详细记录）。
2. 确定在操作的前六个月内所期望的表的行数。
3. 每行的字节数乘以表中的行数，从而确定表的存储需求。

一旦了解了数据库中每个表的存储需求，可将数据库文件的初始容量设置为比所有表所需的磁盘空间总和多 25~50% 的额外的空间（如果可能，可设为 50%）。考虑到对每个表中的数值或所期望的行数估计上的误差，还为索引留下了空间，从而 DBMS 可添加索引以提高数据的访问速度，并给予 DBMS 的临时表的空间，当处理复杂的查询并获得大量的结果时就要创建这种临时表。

可将 FILEGROWTH 选项设置为初始数据库文件容量的 10% 并四舍五入为整数。如果初始数据库文件容量为 25MB，则将 FILEGROWTH 设置为 3MB。请监视数据库文件容量，特别是在开始的几个月的操作中。如果发现数据库文件每月增长超过 10%，就应增加 FILEGROWTH 选项值，从而 DBMS 每月只扩展一次数据库文件容量。

作为一条通用的规则，可将事务处理日志文件的初始尺寸设置为初始数据库文件容量的 25%，并将其 FILEGROWTH 设置为其初始容量的 10%。这样一来，如果初始数据库文件容量为 250MB，可将事务处理日志文件的初始容量设置为 25MB，而其每次增长为 3MB。请监视事务处理日志文件容量的增长并调节其增长因子，从而使 DBMS 每月只扩展一次。

现在读者已经在技巧 24 中学习了如何使用 CREATE DATABASE 语句为数据库和事务处理日志文件设置初始文件容量和增长增量（FILEGROWTH），而在技巧 25 中学习了如何使用 MS-SQL Server Enterprise Explorer。在创建了数据库文件和事务处理日志之后，可通过以下步骤使用 Enterprise Manager 来改变两个文件的容量或其增长因子：

1. 为了启动 Enterprise Manager，单击 Start 按钮，将鼠标指针移动到 Start 菜单中的 Programs 上，选择 Microsoft SQL Server 7.0 并单击 Enterprise Manager。
2. 为了显示 SQL Servers 的列表，单击 SQL Server Group 左边的加号（+）。
3. 为了显示 SQL Server 上资源的列表，包括想要修改的数据库和事务处理日志文件，单击 SQL Server 名左边的加号（+）。例如，如果想要对名为 NVBizNet2 的 SQL Server 加以处理，则单击 NVBizNet2 左边的加号（+）。Enterprise Manager 将显示代表由 SQL Server NVBizNet2（在本例中）所管理的资源的列表。
4. 单击 Databases 文件夹。Enterprise Manager 将在其右窗格中显示 SQL Server 上的数据库。
5. 双击想要修改的数据库和事务处理日志文件数据库图标。对于本例来说，双击 SQLTips（如果在技巧 24 中创建了此数据库）。Enterprise Manager 将显示 SQLTips Properties 对话框的 General 选项卡，这与图 1.19 中所示的类似（对话框名是 <database name> Properties）。因而，仅当双击了 SQLTips 数据库时，对话框才是 SQLTips Properties。
6. 在该对话框中的 Database files 区域中单击 Space Allocated（所分配的空间）字段。对于本例来说，请将 10 改为 15。

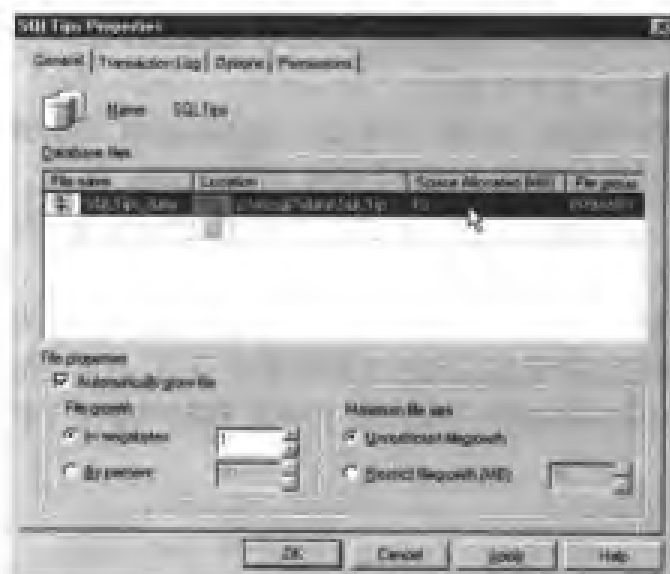


图 1.19 Enterprise Manager 数据库属性对话框

7. 为了使数据库文件的增长数为其当前容量的百分数而不是以 MB 表示的常数, 可在该对话框的 File Properties 区域单击 By Percent 单选钮。对于本例来说, 可保留其默认值 10%。

8. 如果想要限制数据库文件只能增长到某一 MB 数, 可单击 Restrict Filegrowth (MB) 单选钮并键入以 MB 计的最大文件尺寸。对于本例来说, 单击 Unrestricted Filegrowth (不限制文件增长) 单选钮, 从而不限制文件的增长。

9. 单击 Transaction Log 选项卡处理事务处理日志属性。对于本例来说, 不要改变处理事务处理日志属性。但是, 如果确实要改变处理事务处理日志选项, 可按照步骤 6~8 进行, 只要在每步中用 transaction log 代替 database file 即可。

10. 单击 OK 按钮。Enterprise Manager 将把改变施加于 SQLTips 数据库并返回 Enterprise Manager 应用程序窗口。

对于数据库和事务处理日志的最佳初始尺寸和增长增量与数据量、可用的物理存储空间以及期望 DBMS 来处理的事务处理量有关。如果正在从一种 DBMS 产品转向另一种, 可根据文件尺寸和增量的历史需求来决定具体设置。另外, 也可使用本技巧中的数字作为合理的起点。要理解的重要事情是, 虽然不想分配从不需要的空间, 但最好还是让 DBMS 花尽可能少的时间来增加数据库和事务处理日志文件的容量。

技巧 28 理解 MS-SQL Server 的 TempDB 数据库

每当启动 MS-SQL Server 时, DBMS 都创建一个特殊的名为 TempDB 的数据库。服务器把 TempDB 数据库用来放置临时表、临时表数据和临时的用户创建的全局变量。简短地说, TempDB 数据库是系统的中间暂存器。但是, 用户也可以使用此数据库。

使用 TempDB 的好处是用户对 TempDB 对象 (表、视图、索引等) 执行的操作都不记入日志。因而 DBMS 在 TempDB 中操作数据要比在其他数据库操作数据快一些。

在改变数据库对象及其数据值之前 (而不是 TempDB 对象及其数据), DBMS 必须将更新前的 (原始的) 对象结构和值保存在事务处理日志中。这样一来, 对于非 TempDB 数据, 每个数据操作都涉及两种保存操作——保存原始的, 然后保存更新的值。如果做了大量的改变,

则保存原始数据值可增加相当大的开销。但是当使用 TempDB 对象时，DBMS 只要执行一次存储操作——即将更新的值保存到磁盘上。

使用 TempDB 对象的不利的一面是，不能撤消对 TempDB 对象所做的改变。另外每次关闭 DBMS 并重新启动时，TempDB（及其所有对象）都被删除。由此，在 DBMS 重新启动（并重新创建 TempDB）时，任何保存在 TempDB 中的信息都要丢失。因而，从一个会话期到另一会话期，不要依赖存在于 TempDB 中的信息。

可将 TempDB 用作中间暂存器（正如 MS-SQL Server 所做的）来保存临时数据值和表。对于根据多个表总计数据值以便生成总结报告是特别有用的。不用尝试编写选择和总计数据的 SQL 语句，可通过编写总计在临时的 TempDB 表中的数据的查询，然后执行简单的第二个查询即可生成最后的报告。

第 2 章 使用 SQL 数据定义语言 (DDL)

创建数据表和其他数据库对象

技巧 29 使用 CREATE TABLE 语句创建表

不管是创建永久的还是临时的表，都使用同样的 SQL CREATE TABLE 语句，其句法如下：

```
CREATE TABLE <table name>
  (<column definition> [, ...<last column definition>]
   [<primary key definition>]
   [<foreign key definition>])
<column definition> is defined as:
  <column name> <data-type> [DEFAULT <value>]
  [NOT NULL][UNIQUE][<check constraint definition>]
<check constraint definition> is defined as:
  CHECK (<search condition>)
<primary key definition> is defined as:
  PRIMARY KEY (<column name> [, <column name>])
<foreign key definition> is defined as:
  FOREIGN KEY (<column name>) REFERENCES <table name>
```

请参考表 2.1，从中可得到有关 CREATE TABLE 语句的关键字和选项的简要解释。

表 2.1 CREATE TABLE 语句关键字和选项的定义

关键字/选项	描述
table name	表名——在拥有者的数据库内必须是惟一的（参见技巧 6“理解表名”）
column name	列名——在表内必须是惟一的
data-type	SQL 数据类型之一或命名的域（参见技巧 10“理解域”）
DEFAULT <value>	当创建了一行而且并未赋予列以明确的初始值时赋予的一列的值（参见技巧 34“在 CREATE TABLE 语句中使用 DEFAULT 子句设置默认列值”）
NOT NULL	防止向列赋予 NULL 值的约束（参见技巧 143“使用 NOT NULL 列约束防止列中的 NULL 值”）
UNIQUE	防止向 unique 列中添加具有相同值的两个表行的约束（参见技巧 144“使用 UNIQUE 列约束防止列中的重复值”）
CHECK <search condition>	search condition（搜索条件）可为任何求值为 TRUE 或 FALSE 的 SQL 语句。check 约束可防止向 search condition 求值为 FALSE 的表中添加行（参见技巧 145“使用 CHECK 约束确认列值”）
PRIMARY KEY	防止向表中添加两个在某一列（或某几个列）上具有相同值的表行的约束。表可只有一个 PRIMARY KEY。PRIMARY KEY 是可在另一表中被引用为 FOREIGN KEY 的列（或几列）（参见技巧 129“使用 PRIMARY KEY 列约束惟一地确定表行”）

续表

关键字/选项	描述
FOREIGN KEY	其值可作为由 REFERENCES <table name>指定的表中的 PRIMARY KEY 的列（参见技巧 130 “理解引用完整性检查和外键”）

例如，如果执行如下 CREATE TABLE 语句，DBMS 将创建两个表：ITEM_MASTER 和 ORDERS：

```
CREATE TABLE item_master
  (item_number INTEGER,
   description VARCHAR(35) NOT NULL
   PRIMARY KEY (item_number))
CREATE TABLE orders
  (order_number  INTEGER UNIQUE NOT NULL,
   item_number   INTEGER NOT NULL,
   quantity      SMALLINT DEFAULT 1,
   item_cost     DECIMAL (5,2),
   customer_number INTEGER
   PRIMARY KEY (order_number, item_number)
   FOREIGN KEY (item_number) REFERENCES item_master)
```

其中 ITEM_MASTER 表有两个列：DESCRIPTION 和 ITEM_NUMBER。ITEM_NUMBER 是 ITEM_MASTER 表的 PRIMARY KEY，也就是说，表中的每个项目（行）将有惟一的 ITEM_NUMBER 值。换一种说法，没有两个数据项描述有相同的项目号。

第二个表 ORDERS 有 5 个列。ORDERS 表的 PRIMARY KEY 是一个复合的关键字，也就是说，它是由两个或多个列组成的。在本例中，ORDERS 表的 PRIMARY KEY 是由 ORDER_NUMBER 和 ITEM_NUMBER 列组成的，也就是说，相同的 ORDER_NUMBER（如 123）可出现在几个表行上，但具有相同 ORDER_NUMBER 的每一行应有惟一的 ORDER_NUMBER-ITEM_NUMBER 对。

ORDERS 表中的 FOREIGN KEY 约束告诉 DBMS，ORDERS 表中的 ITEM_NUMBER 的值 REFERENCES 引用 ITEM_MASTER 表中的 PRIMARY KEY。这样一来，DBMS 就可采用 ITEM_NUMBER 列中的值并惟一地确定 ITEM_MASTER 表中的单行。另外，如果要插入的行上的 ITEM_NUMBER 值在 ITEM_MASTER 表中不存在的话，则 DBMS 将不允许向 ORDERS 表添加该行。作为对照，如果一行的 ITEM_NUMBER 值存在于 ORDERS 表中的 ITEM_NUMBER 列中，则不能删除 ITEM_MASTER 文件中的那一行（在技巧 130 中读者还将学习有关 FOREIGN KEY 约束的更多知识）。

技巧 30 使用 MS-SQL Server Enterprise Manager 创建表

除了向 MS-SQL Server Query Analyzer 的 Query 窗格中或是在 ISQL（或 OSQL）的准备提示下键入 CREATE TABLE 之外，MS-SQL Server 还有一个可使用的 GUI 工具。为了使用 Enterprise Manager 创建表，请执行以下步骤：

1. 为了启动 Enterprise Manager，可单击 Start 按钮，将鼠标指针移动到 Start 菜单的 Programs 上并选择 Microsoft SQL Server 7.0，然后单击 Enterprise Manager。

2. 为了显示 SQL 服务器的列表, 可单击 SQL Server Group 左边的加号 (+)。
3. 为了显示想要在其上创建表的 SQL Server 上的资源的列表, 可单击 SQL Server 名称左边的加号 (+)。例如, 如果想要在名为 NVBizNet2 的 SQL Server 上工作, 可单击 NVBizNet2 左边的加号 (+)。Enterprise Manager 将显示代表由 SQL Server NVBizNet2 (在本例中) 管理的资源的文件夹列表。
4. 单击 Database 文件夹。Enterprise Manager 将在其右窗格中显示该 SQL Server 上的数据库列表。
5. 右击想要在其中创建表的数据库。对本例来说, 右击 SQLTips 数据库 (如果并未创建 SQLTips 数据库, 可右击 TempDB 数据库)。Enterprise Manager 将显示弹出式菜单。
6. 将鼠标指针移到弹出式菜单中的 New(新建)上, 然后选择 Table(表)。Enterprise Manager 将显示 Choose Name (选择名称) 对话框。
7. 在 Choose Name 对话框中的 Enter a Name for the Table (键入表名) 区域键入表名。对本例来说, 键入 Item_Master, 然后单击 OK 按钮。Enterprise Manager 将显示 SQL Server Enterprise Manager-New Table 窗口, 如图 2.1 所示。

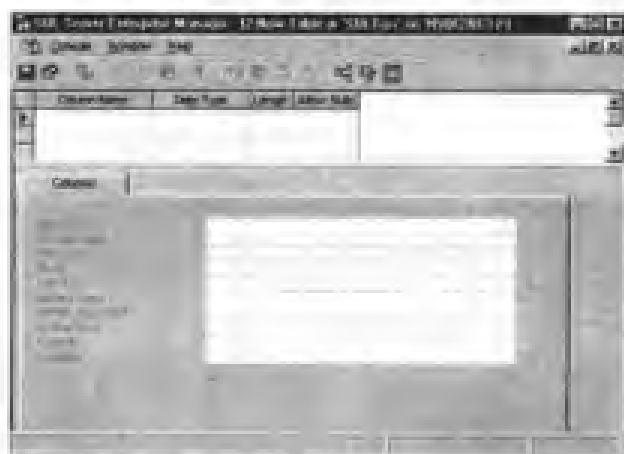


图 2.1 MS-SQL Server Enterprise Manager 的 New Table (新表) 窗口

8. 单击 Column Name 列的第一个单元格准备输入第一列的名称。
9. 键入列名。对本例来说, 键入 item_number, 接着按 Enter 键将输入光标移动到 Datatype 字段。
10. 选择该字段的数据类型。既可单击 Datatype 字段右边的下拉箭头并选择数据类型, 也可向 Data Type 字段键入数据类型。对本例来说, 键入 INT。接着, 按 Enter 键将插入光标移动到 Length 字段上。
11. 如果正在处理字符、图像或文本类型, 可键入字符长度。对本例来说, 处理的是整数, 因而 Length 字段并不适用。按 Enter 键将插入光标移动到 Precision 字段上。
12. 如果正在处理小数或浮点数 (非整数), 可用数字在 Precision 字段内键入总位数。对本例来说, 处理的是整数, 因而精度设置为默认的精度 (这不能改变)。按 Enter 键将插入光标移动到 Scale 字段上。
13. 如果正在处理小数或浮点数 (非整数), 在 Scale 字段内键入想要在小数点右边保留的小数位数。对本例来说, 处理的是整数, 因而 Scale 不适用。按 Enter 键移动到 Allow Nulls

复选框上。

14. 为了允许保存 NULL 值，单击 Allow Nulls 复选框，使选择标记出现。对本例来说，应清除 Allow Nulls 复选框——ITEM_MASTER 表中的每个数据项必须是一个 ITEM_NUMBER。

15. 当向表中插入行时，如果对列未提供明确的值，从而想要把列设置为常数的默认值，可向 Default Value 字段键入该常数值。对本例来说，请把 Default Value 字段留为空白。

16. 当向表中插入行时，如果对列未提供明确的值，为了让 DBMS 为列提供增量值，可单击 Identity 复选框，使之出现选择标记。对本例来说，单击使 Identity 复选框出现选择标记——即想要系统在向 ITEM_MASTER 表添加项目时为新数据项提供项目号。按 Enter 键将插入光标移动到 Seed 字段上。

17. 键入 DBMS 应该提供的第一个值——仅适用于已经将该列标识为具有 IDENTITY 属性的情况。对本例来说，键入 1000。然后按 Enter 键将插入光标移动到 Identity Increment 字段上。

18. 键入当插入新表行时 DBMS 向前一列号上增加的增量值——仅适用于将一列标识为具有 IDENTITY 属性的情况。对本例来说，键入 100。

19. 单击 Column Name 字段中的下一个空白单元格，键入另一列名。

20. 重复步骤 9~步骤 19，直到将表中的所有列都定义完。对本例来说，添加第二个列名为 Description，其数据类型为长度是 35 的 VARCHAR 类型，不允许 NULL 值。Enterprise Manager 将如图 2.2 所示显示表定义。

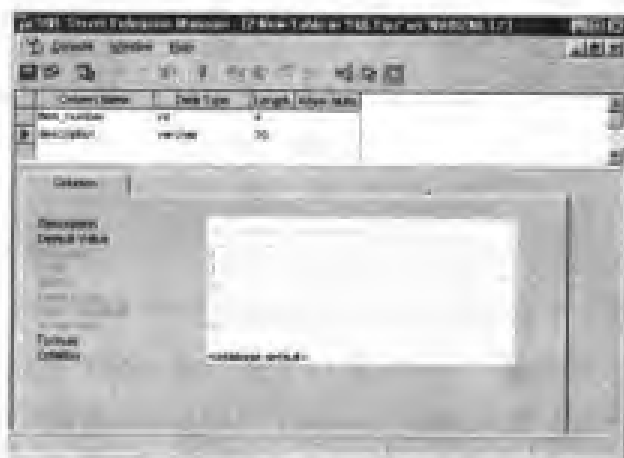


图 2.2 在为 ITEM_MASTER 表定义了两列之后的 MS-SQL Server Enterprise Manager 的 New Table 窗口

21. 为了将一列标识为 PRIMARY KEY，可右击某一列的任意字段并从弹出式菜单中选择 Set Primary Key(设置为 Primary Key)命令。对本例来说，右击 ITEM_NUMBER 上的 Column Name 字段，然后从弹出式菜单中选择 Set Primary Key。

注意：如果想要使用多列（复合的）PRIMARY KEY，可右键单击表中的任意单元格并从弹出式菜单中选择 Properties（属性）命令。Enterprise Manager 将显示 Properties 对话框。单击 Indexes/Keys 选项卡并在 Indexes/Keys 选项卡的 Type 区域内的 Column Name 清单中选择想要包括在 PRIMARY KEY 的列。当完成为 PRIMARY KEY 选择列的任务时，单击 Close 按钮（读者将在技巧 122 “理解 MS-SQL Server 的 CREATE INDEX 语句选项”中学习有关使用 Enterprise Manager 创建索引的更多知识）。

22. 为了保存表定义, 可单击 New Table 标准工具栏上的 Save 按钮 (软盘图标左边的第一个按钮)。

23. 为了关闭 MS-SQL Server Enterprise Manager 的 New Table 窗口, 可单击应用程序窗口右上角的“关闭”按钮 (X)。

可使用 CREATE TABLE 语句 (读者已在技巧 29 “使用 CREATE TABLE 语句创建表”中学习了有关知识) 或 Enterprise Manager GUI 的 New Table 工具来创建 MS-SQL Server 表。不管是 SQL 还是 GUI 都允许定义列、设置约束并确定表键。如果在计算机中装有 MS-SQL Server Enterprise Manager, 选择创建表的方法问题只是一种个人喜好 (命令行或是 GUI)。

技巧 31 创建 MS-SQL Server 的临时表

MS-SQL Server 允许创建两种类型的临时表: 全局的和本地的。本地临时表仅在创建表的会话期间才是可用的, 而且在会话结束时 DBMS 自动地删除临时表。全局临时表对多个数据库会话期间都是可用的。DBMS 在使用临时表的最后一个用户结束其会话时删除全局临时表。

每当登录到数据库时就开始了一个新的会话。因而, 当使用 Query Analyzer 连接到数据库时, 也就开始了一个会话期。当注销或是结束 Query Analyzer 时, DBMS 也结束会话。如果登录到数据库, 每当执行一个存储过程或是运行登录到同一个或另外一个数据库的应用程序时, DBMS 就开始一个新的会话。如果第二次登录到同一数据库, DBMS 保持原来的会话仍然为打开状态, 但在第一次会话创建的临时表中的信息对第二个会话是不可用的。

当需要对同一数据集做几种操作, 如根据多个表中的数据创建总结报告时, 临时表是有用的。通过选择并将所需的原始数据结合到单个表中, 可避免让 DBMS 多次提取并组合数据。除了消除多次选择操作之外, 使用单个临时表可提高执行速度, 因为 MS-SQL Server 从单个表中提取数据比通过引用从多个基表中提取数据要快。

为了创建本地临时表, 可以 # 号开始表名。由此, 执行以下语句将创建一个本地临时表:

```
CREATE TABLE #customer_orders
    (customer_number INTEGER,
     customer_name VARCHAR (35),
     order_date DATETIME,
     amount MONEY)
```

#CUSTOMER_ORDERS 表仅对创建它的人是可访问的。另外, 当用户注销时, DBMS 将自动地删除该表。

如果想要创建一个全局临时表, 可以两个 # 号开始表名。这样, 如果想要创建多个用户 (和会话) 可访问的临时表, 可如下所示在 CREATE TABLE 语句中使用两个 # 号:

```
CREATE TABLE ##customer_orders
    (customer_number INTEGER,
     customer_name VARCHAR (35),
     order_date DATE_TIME,
     amount MONEY)
```

DBMS 将在最后引用该表的用户注销时删除全局临时表。

技巧 32 使用 Transact-SQL 的 CREATE DEFAULT 语句设置列的默认值

MS-SQL Server 允许创建绑定到列的命名的默认值和用户定义的数据类型。当把一个默认

值绑定一个表列时，如果插入一行包括未指定值的列时，DBMS 将为该列提供默认值（不是 NULL）。在 CREATE TABLE 语句之外创建默认值的好处是，可对默认值使用描述性的名称，将同样的默认值应用于同一表或不同表中的多个列中，还可在任何时候改变或是删除默认值。

Transact-SQL 的 CREATE DEFAULT 语句的句法如下：

```
CREATE DEFAULT [<owner name>.]<name of default>  
AS <constant expression>
```

注意：Transact-SQL 由 Microsoft 对标准的 SQL 的添加所组成。没有一个商业的 DBMS 产品完全支持 SQL-92 标准的所有规定。相反，每个开发商都添加了其自己的 SQL 扩展（如 CREATE DEFAULT）并提供编程语言的构建。Microsoft 将其 SQL 及其扩展和编程语言添加称为 Transact-SQL。Oracle 使用 PL/SQL 和 SQL*Plus。虽然大多数标准 SQL-92 代码可在各种 DBMS 产品间移植，但特定的产品扩展（如 Transact-SQL 语句）是不能移植的。如果必须在 Oracle 的 DBMS 中使用 Transact-SQL 语句，请检查一下系统手册。读者很可能发现一个执行同样功能但具有不同的名称句法的 PL/SQL 语句。

所创建的默认值必须符合下列原则：

- 不必为默认值提供<owner name>。但是，如果未提供，则 DBMS 将使用登录名作为默认的<owner name>名。
- 用于默认值的名称（<name of default>）对于拥有者来说必须是惟一的。
- <constant expression>必须只包括常数值，如数字、字符、字符串、内建的函数或是数学表达式。<constant expression>不能包括任何列或其他数据库对象。
- 在创建了一个默认值之后，必须使用存储过程 sp_bindefault 在插入一行时 DBMS 为该列提供值之前将默认值绑定到一列上。
- 默认值必须与绑定其上的列相兼容。例如，如果将字符串绑定到数字列上，DBMS 将生成一条错误信息，而且每当必须为列提供默认值时都不会插入该行。
- 如果为字符列提供字符串默认值而且默认值长于列的长度，则 DBMS 将截短默认值以便能在列中放得下。
- 如果一列既有默认值又有约束，则默认值不能侵犯约束。如果列的默认值侵犯了约束，DBMS 将生成一条错误信息，同时每当必须为列提供默认值时不会插入该行。

例如，假设有一个定义如下的表：

```
CREATE TABLE employee  
(employee_ID INTEGER,  
first_name VARCHAR(20),  
last_name VARCHAR(30),  
social_security_number CHAR(11),  
street_address VARCHAR(35),  
health_card_number CHAR(15),  
sheriff_card_number CHAR(15)  
PRIMARY KEY (employee_ID))
```

而且如果在向 EMPLOYEE 表中添加新雇员时不知道其 Social Security（社会保险号）、health card number（健康卡号）或是 sheriff card number（行政卡号）时想要提供 applied for 和 unknown 来代替 NULL，这时可执行以下的 Transact-SQL 语句创建所需的默认值：

```
CREATE DEFAULT ud_value_unknown AS "Unknown"
```

```
CREATE DEFAULT ud_applied_for AS "Applied for"
```

注意：只能在 Query Analyzer 的 Query 窗格或是在 SOQL (或 OSOQL) 的语句缓冲区中键入一条 CREATE DEFAULT 语句。

在 DBMS 使用默认值之前，必须执行 sp_bindefault 存储过程将该默认值绑定到用户定义的数据类型或是表列上。读者将在技巧 33 中学习如何将默认值绑定到表列上，而在技巧 37 中学习如何将默认值绑定到用户定义的数据类型上。

可以使用存储过程 sp_help 显示用户和系统定义的默认值的列表。由于 sp_help 将显示所有的默认值，而不只是所创建的一个，最好将所有的默认值都组合在一个清单中。为达此目的，可对默认值名使用相同的第一个或是前两个字符（如 UD_，USER DEFAULTS 的简写）。然后，当使用 sp_help 列出数据库的默认值时，该存储过程将把所有创建的默认值都组合在其半字母顺序的所有默认值的清单中。

技巧 33 使用 MS-SQL Server 的存储过程 sp_bindefault 将用户创建的默认值绑定到表列上

正如在技巧 32 “使用 Transact-SQL 的 CREATE DEFAULT 语句设置列的默认值”中所提到的，必须把默认值绑定到表列上。这样一来，DBMS 就能了解哪一列应该设置为哪个默认值。执行存储过程 sp_bindefault 把默认值绑定到表列上的句法如下：

```
EXEC sp_bindefault  
    @DEFNAME=<name of default>,  
    @OBJNAME=<table name>.<column name>
```

其中<name of default>是在 CREATE DEFAULT 语句中给默认值起的名称，而<table name>.<column name>是想要 DBMS 为其提供默认值的表列。

例如，如果执行下面的 Transact-SQL 的 CREATE DEFAULT 语句：

```
CREATE DEFAULT ud_value_unknown AS "Unknown"  
CREATE DEFAULT ud_applied_for AS "Applied for"
```

DBMS 会把默认值 UD_VALUE_UNKNOWN 和 UD_APPLIED_FOR 保存在数据库的系统表中。在创建时，可使用存储过程 sp_bindefault 将默认值绑定到表列上（如在技巧 32 中定义的 EMPLOYEE 表）。

为了将默认值 ud_value_unknown (“Unknown”) 绑定到 EMPLOYEE 表的 SOCIAL_SECURITY_NUMBER 列，可执行以下 Transact-SQL 语句：

```
EXEC sp_bindefault  
    @defname=ud_value_unknown,  
    @objname='employee.[social_security_number]'
```

为了将默认值 ud_applied_for (“Applied For”) 绑定到 EMPLOYEE 表的 SHERIFF_CARD_NUMBER 列，可执行以下 Transact-SQL 语句：

```
EXEC sp_bindefault  
    @defname=ud_applied_for,  
    @objname='employee.[sheriff_card_number]'
```

为了将默认值 ud_applied_for (“Applied For”) 绑定到 EMPLOYEE 表的 HEALTH_CARD_NUMBER 列，可执行以下 Transact-SQL 语句：

```
EXEC sp_bindefault
```

```
@defname=ud_applied_for,
@objname='employee.[health_card_number]'
```

在将默认值绑定到 EMPLOYEE 表列之后，对 EMPLOYEE 表执行以下 INSERT 语句时，DBMS 将为默认值绑定的列提供默认值：

```
INSERT INTO employee (employee_ID, first_name, last_name)
VALUES (1, 'Konrad', 'King')
```

在本例中，DBMS 将为 SOCIAL_SECURITY_NUMBER 列提供默认值“Unknown”，而为 SHERIFF_CARD_NUMBER 和 HEALTH_CARD_NUMBER 列提供默认值“Applied For”，为 STREET_ADDRESS 列提供默认值 NULL。

技巧 34 在 CREATE TABLE 语句中使用 DEFAULT 子句设置默认列值

默认的列值是在未为该列提供一个值时想要 DBMS 为该列输入的字符串或是数字值。读者已经在技巧 32 “使用 Transact-SQL 的 CREATE DEFAULT 语句设置列的默认值”中学习了如何创建默认列值，并在技巧 33 “使用 MS-SQL Server 的存储过程 sp_bindefault 将用户创建的默认值绑定到表列上”中学习了如何将默认值绑定到一个或多个表中的多个列中。遗憾的是，Transact-SQL 的 CREATE DEFAULT 语句以及 sp_bindefault 存储过程仅对工作于 MS-SQL Server 上的用户才是可用的。

标准的 SQL-92 的 CREATE TABLE 语句（在所有的 SQL 关系 DBMS 产品上都是可用的）也给出了在创建表时为列定义默认值的能力。不仅设置默认列值对各种 DBMS 产品都是标准的，而且还比 Transact-SQL 的默认值创建和绑定过程要简单一些。

为了定义列的默认值，只要在 CREATE TABLE 语句中添加 DEFAULT 关键字，后跟列定义的默认值。例如，以下的 SQL CREATE TABLE 语句：

```
CREATE TABLE employee
(employee_ID          INTEGER,
 first_name          VARCHAR(20) NOT NULL,
 last_name           VARCHAR(30) NOT NULL,
 social_security_number CHAR(11) DEFAULT 'Unknown',
 street_address       VARCHAR(35) DEFAULT 'Unknown',
 health_card_number   CHAR(15)  DEFAULT 'Applied For',
 sheriff_card_number   CHAR(15)  DEFAULT 'Applied For',
 hourly_rate          NUMERIC(5,2) DEFAULT 10.00,
 bonus_level          INTEGER    DEFAULT 1,
 job_rating_90days    SMALLINT,
 job_rating_180days   SMALLINT,
 job_rating_1year     SMALLINT
 PRIMARY KEY (employee_ID))
```

为 SOCIAL_SECURITY_NUMBER、STREET_ADDRESS、HEALTH_CARD_NUMBER、SHERIFF_CARD_NUMBER、HOURLY_RATE 和 BONUS_LEVEL 定义了默认值。因而，当执行 SQL 的以下 INSERT 语句时：

```
INSERT INTO employee
(employee_ID, first_name, last_name,
social_security_number, street_address)
```

```
VALUES (1, 'Konrad', 'King', NULL, '77 Sunset Strip')
```

DBMS 将把 HEALTH_CARD_NUMBER 和 SHERIFF_CARD_NUMBER 设置为“Applied For”，将 HOURLY_RATE 列设置为 10.00，而把 BONUS_LEVEL 列设置为 1。虽然 SOCIAL_SECURITY_NUMBER 和 STREET_ADDRESS 列有默认值，但由于 INSERT 语句将 SOCIAL_SECURITY_NUMBER 设置为 NULL 而将 STREET_ADDRESS 列设置为“77 Sunset Strip”，所以其默认值都未使用。在本例中，确实没有为 3 个工作工资水平列 (JOB_RATING_90DAYS、JOB_RATING_180DAYS 和 JOB_RATING_1YEAR) 定义默认值，所以 DBMS 将这些列都设置为 NULL。

注意：在 CREATE TABLE 语句设置列默认值之前，请检查一下，自己的 DBMS 在创建了表之后是允许改变默认值还是停止使用默认值。MS-SQL Server 不允许使用 ALTER TABLE 语句来添加、改变或是删除任何在 CREATE TABLE 语句中定义的列默认值。（可使用 ALTER TABLE 语句来添加新列并向该列赋予默认值。但是，一旦列成为表的一部分，就不能改变其默认值。）如果正在使用 MS-SQL Server，可以使用 Transact-SQL 的 CREATE DEFAULT 语句在表定义之外创建一个命名的列默认对象，从而绕过此不足之处。MS-SQL Server 允许使用 sp_bindefault 存储过程将一个命名的列默认值绑定到一列上，可在任何时候通过将默认值与所有列解除绑定、将其删除，再重新用新值创建列默认值，然后再重新将其绑定到一个或多个表的列上，从而改变命名列默认值。

技巧 35 使用 MS-SQL Server Enterprise Manager 为用户定义的数据类型或表列创建默认值

像通常一样，MS-SQL Server 既有命令行 Transact-SQL 语句也有使用 MS-SQL Server Enterprise Manager 的 GUI 方法来创建默认值（读者已经在技巧 32 “使用 Transact-SQL 的 CREATE DEFAULT 语句设置列的默认值”中学习了有关知识）。在 CREATE TABLE 语句之外创建默认值的好处是，可为默认值起一个有意义的名称，将其用于用户定义的数据类型或是在一个表或多个表中的多个列中，可在任何时候改变默认值或是完全停止使用该默认值。

为了使用 MS-SQL Server Enterprise Manager 创建默认值，请执行以下步骤：

1. 为了启动 Enterprise Manager，单击 Start 按钮，将鼠标指针移动到 Start 菜单的 Programs 上，选择 Microsoft SQL Server 7.0，然后单击 Enterprise Manager。
2. 为了显示 SQL 服务器的列表，单击 SQL Server Group 左边的加号 (+)。
3. 为了显示带有要在其中创建默认值的数据库的 SQL Server 上的资源的列表，可单击 SQL Server 名称左边的加号 (+)。例如，如果想要对名为 NVBizNet2 的 SQL Server 加以处理，可单击 NVBizNet2 左边的加号 (+)。Enterprise Manager 将显示代表由 NVBizNet2（在本例中）管理的资源的文件夹。
4. 单击 Databases 文件夹。Enterprise Manager 将在其右窗格中显示此 SQL Server 上的数据库。
5. 单击想在其中创建默认值的数据库。对本例来说，单击 SQLTips（如果在技巧 24 中创建了此数据库，如果未创建 SQLTips 数据库，单击 Northwind 示例数据库）。
6. 选择 Action 菜单中的 New 选项，单击 Default。Enterprise Manager 将显示 Default Properties（默认属性）对话框，如图 2.3 所示。

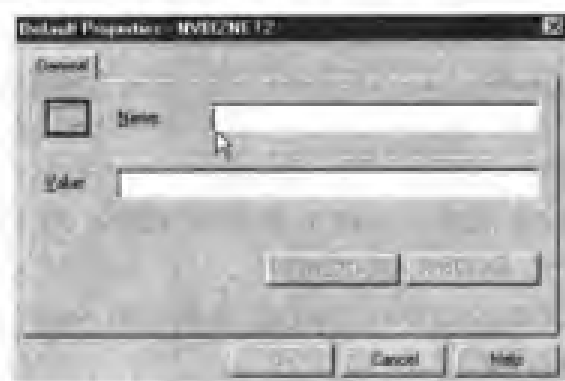


图 2.3 Enterprise Manager 的 Default Properties 对话框

7. 在 Name 区域中键入默认值名，对本例来说，键入 `ud_minimum_wage`，然后按 Tab 键。
8. 在 Value 区域中键入默认值。既可键入数字也可键入字符串，内建函数或是数学表达式。值不能包括任何列或其他数据库对象。对本例来说，可在 Value 区域中键入 7.35。
9. 单击 OK 按钮。Enterprise Manager 将把 `UD_MINIMUM_WAGE` 默认值定义添加到系统表中并关闭 Default Properties 对话框。

正如读者在技巧 34 “在 CREATE TABLE 语句中使用 DEFAULT 子句设置默认列值”中所学的，必须将默认值绑定到一列或用户定义的数据类型上，以便 DBMS 实际使用所创建的默认值。技巧 34 中，使用了存储过程 `sp_bindefault` 来把默认值绑定到表列上。在技巧 37 “使用 MS-SQL Server Enterprise Manager 将默认值绑定到数据类型或表列”中，读者将学习如何使用 Enterprise Manager 将默认值绑定到数据类型或表列上。

技巧 36 使用 MS-SQL Server Enterprise Manager 创建用户定义的数据类型

读者已在技巧 29 “使用 CREATE TABLE 语句创建表”中学习了创建表时如何使用数据类型。正如读者已经知道的，每个表列必须有一种定义的可向该列输入数据的数据类型。例如，如果某列是 INTEGER 类型，则该列只可存储整数——字符和带小数的数字是不允许的。类似地，将某列定义为 CHAR(10) 类型，则该列可存储多达 10 个字符、符号或是数字。

用户定义的数据类型允许使用标准 SQL 数据类型之一或已经创建的域来为一列中可找到的数据类型定义一个描述性的名称，将其定义为该种（用户定义的）数据类型。例如，假设正在处理 EMPLOYEE 表中的 REGULAR_PAY_RATE 列，可将该列的数据类型定义为 NUMERIC(5,2)，或者可使用更有描述性的用户定义的数据类型，如 HOURLY_PAY_RATE。

为了使用 Enterprise Manager 创建一个用户定义的数据类型，可执行以下步骤：

1. 为了启动 Enterprise Manager，单击 Start 按钮，将鼠标指针移动到 Start 菜单的 Programs 上，选择 Microsoft SQL Server 7.0，然后单击 Enterprise Manager。
2. 为了显示 SQL 服务器的列表，单击 SQL Server Group 左边的加号 (+)。
3. 为了显示带有要在其中创建用户定义的数据类型的数据库的 SQL Server 上的资源的列表，可单击 SQL Server 名称左边的加号 (+)。例如，如果想要对名为 NVBizNet2 的 SQL Server 加以处理，可单击 NVBizNet2 左边的加号 (+)。Enterprise Manager 将显示代表由 NVBizNet2（在本例中）管理的资源的文件夹。
4. 单击 Databases 文件夹。Enterprise Manager 将展开服务器列表以便展示在步骤 3 中选

择的 SQL Server 上的数据库列表。

5. 单击想在其中创建数据类型的数据库左边的加号 (+)。对本例来说, 单击 SQLTips (如果在技巧 24 “使用 CREATE DATABASE 语句创建 MS-SQL Server 数据库和事务处理日志”中创建了此数据库) 左边的加号 (+)。如果未创建 SQLTips 数据库, 单击 Northwind 示例数据库左边的加号 (+)。Enterprise Manager 将显示数据库对象类型的清单。

6. 单击 User-Defined Data Types (用户定义的数据类型)。Enterprise Manager 将在应用程序窗口的右窗格中显示已有的用户定义的数据类型。

7. 选择 Action 菜单中的 New User-Defined Data Type (新建用户定义的数据类型) 选项。Enterprise Manager 将显示 User-Defined Data Type Properties (用户定义的数据类型属性) 对话框, 如图 2.4 所示。

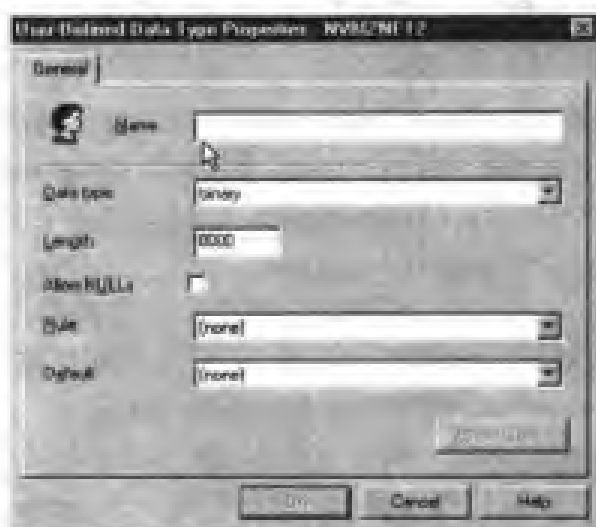


图 2.4 Enterprise Manager 的 User-Defined Data Type Properties 对话框

8. 在 Name 区域键入数据类型的名称。对本例来说, 键入 hourly_pay_rate。

9. 单击 Data Type 区域右边的下拉列表按钮列出可用的 SQL 数据类型并为用户定义的数据类型选择一个数据类型 (实际上并未创建一种“新”数据类型, 而只是为已有的数据类型起了一个具有描述性名称)。对本例来说, 选择 money。

10. 如果想要对用户定义的数据类型的该列允许 NULL 值, 单击 All NULLS 复选框使之出现选择标记——如果雇员是有薪水的或是只按委任来支付工资, 则想要对每小时所付工资列允许 NULL 值。

11. 如果想要使用一种数据库规则来施加一种约束, 以限制用户可向定义为正在定义的数据类型的列中输入的值, 可使用 Rule 区域右边的下拉列表按钮, 显示数据库规则的清单并选择一种规则 (读者将在技巧 147 “使用 Transact-SQL 的 CREATE RULE 语句创建 MS-SQL Server 规则”中学习如何创建规则)。对本例来说, 选择 (none)。

12. 如果在用户插入包括定义为用户定义的数据类型的列的行时未提供值, 如想让 DBMS 提供默认值, 则可使用 Default Name 区域右边的下拉列表按钮来显示已定义的默认值的清单。对本例来说, 选择 (none)。

13. 单击 OK 按钮。

在完成了步骤 13 之后, Enterprise Manager 将把数据类型定义保存在 DBMS 的系统表中。

可在数据库中的任何使用标准的 SQL 数据类型的地方使用自己定义的数据类型。对本例来说，只要创建了 HOURLY_PAY_RATE 数据类型，以下 SQL 语句就是合法的：

```
CREATE TABLE employee
(
    id      INTEGER,
    name    VARCHAR(35),
    regular_pay_rate hourly_pay_rate)

```

注意：数据库中用户定义的数据类型名必须对拥有者来说是惟一的，且必须是在想要使用该用户定义的数据类型名的数据库中定义。例如，如果在 SQLTips 数据库中定义了 HOURLY_PAY_RATE，而且想要在 SQLTips 数据库表以及 Northwind 数据库表中使用 HOURLY_PAY_RATE，则其也必须在 Northwind 数据库中进行定义。

技巧 37 使用 MS-SQL Server Enterprise Manager 将默认值绑定到数据类型或表列

在 MS-SQL Server 使用所创建的默认值之前，必须将默认值绑定到表列或一个用户定义的数据类型上。在技巧 33 “使用 MS-SQL Server 的存储过程 sp_bindefault 将用户创建的默认值绑定到表列上”中读者学习了如何使用存储过程 sp_bindefault 将默认值绑定到表列上。在本技巧中，读者将学习如何使用 Enterprise Manager。

为了使用 Enterprise Manager 将默认值绑定到表列上，请执行以下步骤：

1. 为了启动 Enterprise Manager，单击 Start 按钮，将鼠标指针移动到 Start 菜单的 Programs 上，选择 Microsoft SQL Server 7.0，然后单击 Enterprise Manager。
2. 为了显示 SQL 服务器的列表，单击 SQL Server Group 左边的加号 (+)。
3. 为了显示要在其中将默认值绑定到列的数据库的 SQL Server 上的资源的列表，可单击 SQL Server 名称左边的加号 (+)。例如，如果要对名为 NVBizNet2 的 SQL Server 加以处理，可单击 NVBizNet2 左边的加号 (+)。Enterprise Manager 将显示代表由 NVBizNet2（在本例中）管理的资源的文件夹。
4. 单击 Databases 文件夹左边的加号 (+)。Enterprise Manager 将展开 SQL Server 的数据库分支以显示在步骤 3 中选择的 SQL Server 上的数据库列表。
5. 单击想在其中绑定默认值的数据库左边的加号 (+)。对本例来说，单击 SQLTips（如果在技巧 24 “使用 CREATE DATABASE 语句创建 MS-SQL Server 数据库和事务处理日志”中创建了此数据库）左边的加号 (+)，如果未创建 SQLTips 数据库，单击 Northwind 示例数据库左边的加号 (+)。Enterprise Manager 将显示数据库对象类型的清单。
6. 在数据库对象类型的清单上单击 Defaults 图标 Enterprise Manager 将使用其右窗格显示在步骤 5 中选择的数据库中的用户定义的默认值，如图 2.5 所示。
7. 双击想要绑定到列的默认值名，对本例来说，双击 ud_minimum_wage（如果在技巧 35 “使用 MS-SQL Server Enterprise Manager 为用户定义的数据类型或表列创建默认值”中创建了 UD_MINIMUM_WAGE 默认值）。Enterprise Manager 将显示 Default Properties 对话框，如图 2.6 所示。
8. 单击 Bind Columns（绑定列）按钮。Enterprise Manager 将显示 Bind Default to Columns（将默认值绑定到列）对话框，如图 2.7 所示。

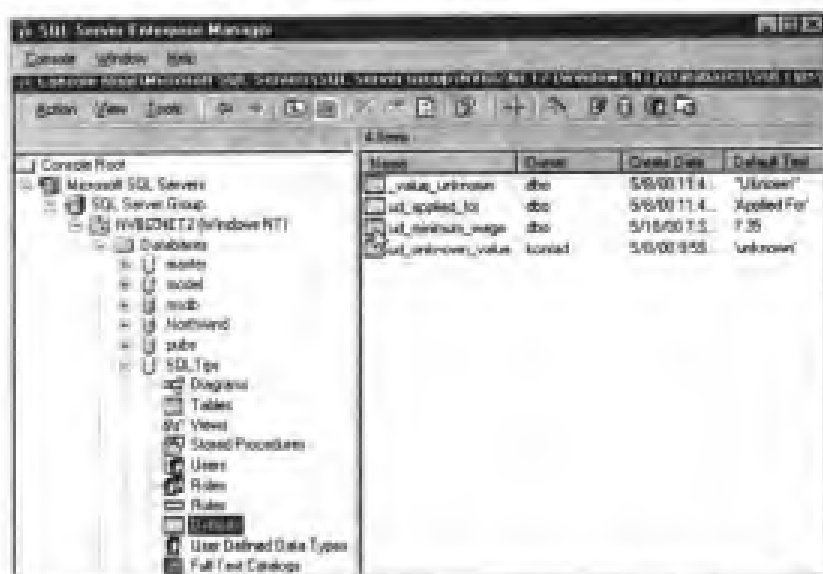


图 2.5 显示数据库中用户定义的默认值的 Enterprise Manager 应用程序窗口



图 2.6 在选择了 UD MINIMUM WAGE 默认值之后的 Default Properties 对话框



图 2.7 Bind Default to Columns 对话框

9. 单击 Table 区域右边的下拉列表按钮以显示数据库中表的清单。单击带有要将默认值绑定其上的列的表。对本例来说, 单击 EMPLOYEE 表。Enterprise Manager 将在 Unbound (未

绑定）列清单中显示表列的清单。

10. 在 Bind Default to Columns 对话框底部的 Unbound（未绑定）列清单中找出要将默认值绑定其上的列并单击。对本例来说，单击 REGULAR_PAY_RATE。

11. 单击 ADD 按钮。Enterprise Manager 将把 REGULAR_PAY_RATE 列添加到 Bound（已绑定）列清单中（由于左边的列表只显示未绑定列，读者将发现 Enterprise Manager 在将 REGULAR_PAY_RATE 列放入 Bound 列清单时，就从 Unbound 列清单中删除了 REGULAR_PAY_RATE 列）。

12. 单击 OK 按钮。Enterprise Manager 将返回 Default Properties 对话框。

在完成了步骤 12 之后，Enterprise Manager 将在系统表中记下默认值绑定。当用户向表中添加行而未为绑定的列指定值时，DBMS 就接着把该列设置为默认值。

如果以后决定不再想让 DBMS 为特定的列提供默认值，可执行存储过程 sp_unbindefault 或执行将默认值绑定到列的步骤 1~步骤 9。然后在步骤 10 中不是选择一个未绑定的列，而是在 Bind Default to Columns 对话框的底部从 Bound 列清单中选择想要解除绑定的列。接着在步骤 11 中，单击 Remove 按钮。最后，在步骤 12 中单击 OK 解除默认值的绑定。

除了将默认值绑定到列之外，还可将默认值绑定到用户定义的数据类型。一旦这样做了，当一个用户未为定义为某一数据类型的表列且已绑定了默认值的表列提供值时，DBMS 将提供默认值而不是 NULL。

例如，为了将 UD_MINIMUM_WAGE 默认值绑定到在技巧 36 “使用 MS-SQL Server Enterprise Manager 创建用户定义的数据类型”中定义的 HOURLY_PAY_RATE 数据类型，可执行以下步骤：

注意：如果在此技巧开始时退出了 Default Properties 对话框或是未执行绑定默认值过程，如有必要可执行绑定默认值过程中的步骤 1~步骤 7，以便显示 Default Properties 对话框。

1. 在 Default Properties 对话框中单击 Bind UDTs 按钮。Enterprise Manager 将显示在 Bind Default to User-Defined Data Types（将默认值绑定到用户定义的数据类型）对话框中定义的默认值，如图 2.8 所示。



图 2.8 Bind Default to User-Defined Data Types 对话框

2. 找出想要绑定默认值的数据类型, 单击 Bind 中的复选框使之出现选择标记。对于本例来说, 单击 HOURLY_PAY_RATE 数据类型右边的复选框 (如果在技巧 36 中创建了的话)。

3. 如果想要把默认值只绑定到想要绑定默认值的用户定义的数据类型的将来的列中, 单击 Future Only 复选框使之出现选择标记。——DBMS 以后将把默认值既用作已经声明为 HOURLY_PAY_RATE 数据类型的列, 也用于那些将来要定义为该种数据类型的列。

4. 单击 OK 按钮。Enterprise Manager 将在 DBMS 系统表中记下默认值的绑定并返回 Default Properties 对话框。

如果今后决定不再想让 DBMS 为那些定义为特定的用户定义的数据类型且已绑定了默认值的列提供默认值, 则可执行存储过程 sp_unbinddefault, 或是返回 User-Defined Data Types 对话框并为该用户定义的数据类型清除 Bind 复选框。

当从数据类型上解除绑定默认值时, 必须确定是否想要把默认值仍然绑定到已有的数据类型列上。如果单击了 Future Only 复选框上的选择标记, DBMS 会继续为已有的用户定义的数据类型列提供默认值。如果清除了 Future Only 复选框, 则 DBMS 将为已有的数据类型列和将来创建的任何列都提供 NULL 值 (停止提供默认值)。

为了返回 Enterprise Manager 应用程序窗口, 可在 Default Properties 对话框中单击 OK 按钮。

技巧 38 使用 Transact-SQL 的 DROP DEFAULT 语句从数据库中删除默认值

当不再需要已经创建的默认值时, 可使用 Transact-SQL DROP DEFAULT 语句永久地从创建默认值的数据库中删除默认值。DROP DEFAULT 语句的句法如下:

```
DROP DEFAULT <default name>
[, <default name>, ... ,<last default name>]
```

其中<default name>是定义默认值时为其起的名称。正如从此语句的句法中所看到的, 可用一条 DROP DEFAULT 语句删除多个默认值。

在技巧 32 “使用 Transact-SQL 的 CREATE DEFAULT 语句设置列的默认值”中, 曾经使用以下 Transact-SQL 语句创建默认值 UD_VALUE_UNKNOWN 和 UD_APPLIED_FOR:

```
CREATE DEFAULT ud_value_unknown AS "Unknown"
CREATE DEFAULT ud_applied_for AS "Applied for"
```

如果不再需要这些默认值, 可执行以下 Transact-SQL 语句将其从数据库中删除:

```
DROP DEFAULT ud_value_unknown, ud_applied_for
```

要了解的一件重要事情是, 不能删除当前已绑定到列或用户定义数据类型的默认值。必须首先使用 Enterprise Manager 或存储过程 sp_unbinddefault 解除与所有列和用户定义的数据类型的绑定。在安全解除默认值的绑定之后, 即可将其从数据库中删除。

如果创建并使用默认值, 保留默认值和 (或许更为重要的) 默认值绑定其上的列或用户定义的数据类型的准确清单是重要的。遗憾的是, 如果在试图删除带有绑定的默认值状态的列时, DBMS 返回一条错误而且不能识别引起错误的绑定。例如, 如果执行以下语句, 而且 UD_APPLIED_FOR 与列或数据类型绑定:

```
DROP DEFAULT ud_applied_for
```

则此语句将不能成功执行, DBMS 显示以下错误信息:

```
The default 'ud_applied_for' cannot be dropped because it
is bound to one or more column.
```

为了成功地执行 DROP DEFAULT 语句，必须运行存储过程 sp_unbindefault。但是，sp_unbindefault 需要用户提供表和列名，才能解除默认值的绑定。另外，MS-SQL Server 并不能列出绑定有默认值的列的存储过程，因而必须参考文档。

如果没有想要删除的默认值绑定的清单，或是如果喜欢使用 GUI 工具，可使用 MS-SQL Server Enterprise Manager 来解除绑定并删除默认值。首先，执行技巧 37 “使用 MS-SQL Server Enterprise Manager 将默认值绑定到数据类型或表列”中绑定默认值过程的前 8 步。在完成步骤 8 之后，DBMS 将显示 Bind Default to Columns（将默认值绑定到列）对话框（请参考技巧 37 中的图 2.7）。

在屏幕上有了 Bind Default to Columns 对话框之后，可执行以下步骤解除默认值的绑定：

1. 为了找出带有一个或多个与想要删除的默认值绑定的表，可单击 Table 区域右边的下拉列表框。Enterprise Manager 将在下拉列表框中显示数据库表的清单。

2. 如果知道想要的表名，将其从下拉列表中选择（如果不知道想要的表，则必须一次选择一个数据库表）。Enterprise Manager 将在 Bound Columns（绑定的列）清单中显示与默认值绑定的表列。

3. 为了解除默认值与列的绑定，在 Bound Columns 清单中单击列名，然后单击 Remove 按钮。

4. 重复步骤 3，直到从 Bound Columns 清单中删除所有的列名。

5. 如果文档列出了其他默认值绑定其上的表（或者没有文档而且也未通过数据库表制作这一文档），单击 Apply 按钮，然后继续步骤 3，从而选择想要处理的下一个表。

6. 当完成了从表列中解除默认值的绑定时，单击 OK 按钮。Enterprise Manager 将返回 Default Properties 对话框（与技巧 37 中的图 2.6 类似）。

7. 单击 OK 按钮。Enterprise Manager 将返回其应用程序窗口，默认值显示在右窗格中，此时类似于技巧 37 中的图 2.5。

既然已经解除了默认值的绑定，现在就可执行以下步骤来删除默认值：

1. 为了选择想要删除的默认值，可在 Enterprise Manager 应用程序的右窗格中默认值列表上单击默认值名。

2. 为了删除默认值，可单击标准工具栏上的 Delete 按钮（带有红色的 X），或选择 Action 菜单上的 Delete 选项。Enterprise Manager 将在 Drop Objects（删除对象）对话框中显示所选的默认值的名称、拥有者和类型。

3. 单击 Drop All（全部删除）按钮。

为退出 Enterprise Manager，可单击应用程序窗口上的关闭按钮（右上角的 X），或选择 Console 菜单上的 Exit 选项。

SQL 的 ALTER TABLE 语句允许做以下事情：

- 向表中添加列
- 从表中删除列
- 改变或删除列的默认值
- 添加或删除单独的列约束
- 改变列的数据类型
- 添加或删除表的主键

- 添加或删除外键
- 添加或删除表的检查约束

ALTER TABLE 语句的句法如下:

```
ALTER TABLE <table name>
    {ADD <column definition>
    |[WITH CHECK | WITH NO CHECK] ADD <table constraint>
    |ALTER COLUMN <column name> <new data type>
    |[(precision,scale)][NULL | NOT NULL]}
    {DROP COLUMN <column name>
    |[,<column name>...,<last column name>]}
    {DROP [CONSTRAINT] <constraint name>}
    {CHECK | NO CHECK CONSTRAINT [ALL |
    <constraint name>
    |,<constraint name>...,<last constraint name>]}
    {ENABLE | DISABLE TRIGGER [ALL |
    <trigger name>
    |,<trigger name>...,<last trigger name>]}
<column definition> is defined as:
    <column name> <data type>
    [IDENTITY [(seed,increment)]|[NOT NULL]
    [DEFAULT <value>]][UNIQUE]
    [<check constraint definition>]
<check constraint definition> is defined as:
    CHECK (<search condition>)
<table constraint> is defined as:
    [CONSTRAINT <constraint name>]
    <primary key definition>
    | <foreign key definition>
    | DEFAULT <constant expression>
    FOR <column name>
    | CHECK (<search condition>)
<primary key definition> is defined as:
    PRIMARY KEY (<column name> [, <column name>])
<foreign key definition> is defined as:
    FOREIGN KEY (<column name>) REFERENCES <table name>
```

注意: 在 ALTER TABLE 语句中不要包括大括号 ({}). 大括号不过是 ALTER TABLE 的不同形式的分隔符, 用户必须为每条语句选择一种 (只能一种) 形式。因而

ALTER TABLE <table name> ADD <column definition>
是一种合法的选择, 正如以下语句一样:

```
ALTER COLUMN <column name> <new data type>
[(precision,scale)][NULL | NOT NULL]
```

而且, 不要在 ALTER TABLE 语句中放管道符 (|)。管道符表示“或”, 其意义为, 必须选择一个子句或另一子句。由此, 当指定一种表约束时, 既可定义主键、外键、默认值, 也可以定义检查约束, 但不能是所有的 4 种都有。

SQL ALTER TABLE 语句的子句的趋势与具体的 DBMS 有关。所有产品都允许添加列。但并不是所有的产品都允许删除列（使用 ALTER 语句删除列并不是 SQL-92 规范中的内容）。某些产品允许添加并删除已有列上的单独列约束，另一些产品包括 MS-SQL Server 却不允许这样做。

简短地说，每一种 DBMS 开发商都向 ALTER TABLE 语句添加他们认为是重要的特色，而取消那些用其他与开发商有关的结构易于实现的子句，例如，不允许使用 ALTER TABLE 语句来改变已有列上的默认值或是向已有列上添加默认值，即使其他开发商允许这样做。而 MS-SQL Server 除了 ALTER TABLE 语句之外，还通过 Transact-SQL 和存储过程提供了 CREATE DEFAULT、sp_bindefault、sp_unbindefault 和 DROP DEFAULT 来管理列默认值。

技巧 39 使用 ALTER TABLE 语句向表中添加列

向表中添加列或许是最常用的 ALTER TABLE 语句形式。添加列的 ALTER TABLE 的句法如下：

```
ALTER TABLE <table name>
    ADD <column name> <data type> [DEFAULT <value>]
    [NOT NULL][IDENTITY][UNIQUE][CHECK (<search condition>)]
```

例如，为了向定义如下的 EMPLOYEE 表中添 BADGE_NUMBER 列：

```
CREATE TABLE employee
(
    employee_id          INTEGER,
    first_name           VARCHAR(20),
    last_name            VARCHAR(30),
    social_security_number CHAR(11),
    street_address       VARCHAR(35)
    PRIMARY KEY (employee_id))
```

可使用以下 ALTER TABLE 语句：

```
ALTER TABLE employee ADD badge_number INTEGER IDENTITY
```

DBMS 将向 EMPLOYEE 表中添加新列 BADGE NUMBER。除此之外（仅对 MS-SQL Server），本句中的 IDENTITY 特性将使 MSSQL Server 设置每行中的 BADGE_NUMBER 列，以 1 开始每加一行加 1。

当使用 ALTER TABLE 语句向表中添加新列时，DBMS 向表的列定义的尾部添加列，而在接下来的查询中将出现在最右边。除非指定默认值（或在 MS-SQL Server 上使用 IDENTITY 约束），DBMS 为已有行上的新列假设 NULL 值。

由于 DBMS 为已有行上的新列假设 NULL 值，当使用 ALTER TABLE 语句添加列时，不能简单地添加 NOT NULL 约束，还必须提供默认值。毕竟，如果未提供默认值，DBMS 假设已有行上的新列为 NULL 值，这就立即侵犯了 NOT NULL 约束。

当向表中添加列时，DBMS 并不实际地扩展已有的行，而是只扩展其表的描述，使之在系统表中包括新列。每当要求 DBMS 读取已有的行时，在呈现查询结果之前，就为新列添加一个（或多个）NULL 值。当 DBMS 更新时，将向新行和已有的行添加新列。

技巧 40 使用 MS-SQL Server 的 ALTER TABLE、DROP COLUMN 子句删除表列

SQL-92 标准并未作为 ALTER TABLE 语句的一部分而指定 DROP COLUMN 子句。因而，

某些 DBMS 产品需要从表中卸载数据, 再使用 DROP TABLE 语句删除表, 执行 CREATE TABLE 语句重新创建没有想要删除的列的表, 然后重新装载在删除表之前卸载的数据 (如果给出所涉及的步骤, 读者可能试图忽略想要删除的列)。

幸运的是, MS-SQL Server 提供了 DROP COLUMN 子句作为其 ALTER TABLE 语句的一部分。删除列的句法如下:

```
ALTER TABLE <table name>
    DROP COLUMN <column name>
    [, <column name> ..., <last column name>]
```

这样一来, 如果 EMPLOYEE 表定义如下:

```
CREATE TABLE employee
    (employee_id INTEGER,
    first_name VARCHAR(20),
    last_name VARCHAR(30),
    social_security_number CHAR(11),
    street_address VARCHAR(35),
    health_card_number CHAR(15),
    sheriff_card_number CHAR(15),
    badge_number IDENTITY(100,100)
    PRIMARY KEY (employee_id))
```

可使用以下 ALTER TABLE 语句从中删除一些列:

```
ALTER TABLE employee
    DROP COLUMN health_card_number, sheriff_card_number
```

以上语句删除了 HEALTH_CARD_NUMBER 和 SHERIFF_CARD_NUMBER 列。

MS-SQL Server 将防止删除那些赋予约束或默认值的列。另外, 不能删除在另外的表中标识为 FOREIGN KEY 的列。例如, 如果试图从 EMPLOYEE 表中删除 EMPLOYEE_ID 列, MS-SQL Server 将用以下错误信息作为响应:

```
Server: Msg 4922, Level 16, State 3, Line 1
ALTER TABLE DROP COLUMN employee_id failed because PRIMARY
KEY CONSTRAINT PK__employee__6E01572D access this column
```

而且 ALTER TABLE 不能成功完成其任务。

为了删除带有默认值或约束的列, 必须首先以如下形式使用 ALTER TABLE 语句删除约束:

```
ALTER TABLE <table name> DROP CONSTRAINT <constraint name>
```

在本例中, CREATE TABLE 语句并未对 EMPLOYEE 表的 EMPLOYEE_ID 列指定 PRIMARY KEY 约束名。因而, DBMS 为 PRIMARY KEY 约束创建惟一的约束名 PK__employee__6E01572D。因而, 为了在本例中从 EMPLOYEE 表中删除 PRIMARY KEY 约束, 可使用以下 Transact-SQL 语句:

```
ALTER TABLE employee DROP CONSTRAINT PK__employee__6E01572D
```

注意: DBMS 赋予未命名约束的名称在每次创建约束时都不相同, 即使删除并重新指定同一约束, 其名称也是不同的。而且, 当使用 DBMS 生成的名称来删除约束时, 一定注意在约束名中使用双下划线 (__) , 而不是单下划线。

如果想要删除在另一表中是 FOREIGN KEY 的列, 必须首先使用 ALTER TABLE 在另一表中删除 FOREIGN KEY 引用, 然后删除当前表中的列 (读者将在技巧 42 中学习使用 ALTER

TABLE 语句来处理 FOREIGN KEY 约束的有关知识)。

技巧 41 使用 ALTER TABLE 语句改变列的宽度或数据类型

与许多 DBMS 产品不同, MS-SQL Server 不仅允许改变列的宽度, 而且还允许改变其数据类型。如果列为以下情况则不能改变其数据类型:

- 是 TEXT、IMAGE、NTEXT 或 TIMESTAMP 类型
- 是索引的一部分, 除非原始数据类型是 VARCHAR 或 VARBINARY, 而且未改变原始数据类型或是使列宽变窄
- 是 PRIMARY KEY 或 FOREIGN KEY 的一部分
- 是用于 CHECK 约束的
- 是用于 UNIQUE 约束中的
- 具有与之联系的默认值
- 是重复的
- 是计算的或用在计算的列中

当改变列的数据类型时, 列中所有已有的数据必须与新的数据类型兼容。因而, 一定要从 INTEGER 转变为字符, 因为 CHARACTER 列可保存数字、字母和特殊符号。但是, 当将 CHARACTER 列转变为 INTEGER 时, 必须保证表的每行上所转变的 CHARACTER 字段具有数字或 NULL 值。

当已经把某列确定为可改变类型的列时, 可使用以下形式的 ALTER TABLE 语句改变列的宽度或数据类型:

```
ALTER TABLE <table name>  
ALTER COLUMN <column name> <new data type>
```

例如, 如果有如下定义的 EMPLOYEE 表:

```
CREATE TABLE employee  
(employee_id INTEGER,  
first_name VARCHAR(20),  
last_name VARCHAR(30),  
social_security_number CHAR(11),  
street_address CHAR(30),  
health_card_number CHAR(15),  
sheriff_card_number CHAR(15),  
PRIMARY KEY (employee_id))
```

可使用以下 ALTER TABLE 语句将 STREET_ADDRESS 列从 CHAR(30) 改变为 CHAR(35):

```
ALTER TABLE employee  
ALTER COLUMN street_address CHAR(35)
```

还可以使用 ALTER TABLE 语句来改变列的数据类型。例如, 将 HEALTH_CARD_NUMBER 列从 CHAR(15) 改变为 INTEGER, 可使用以下 ALTER TABLE 语句:

```
ALTER TABLE employee  
ALTER COLUMN health_card_number INTEGER
```

当把一列从一种数据类型转换为另一种时, 请记住, 列中所有已有的数据必须与新数据类型兼容。因而, 仅当所有当前的健康卡号为 NULL 或是当所有的都是由数字组成时, 将

HEALTH_CARD_NUMBER 列从字符转化为整数是可以的。由此, 如果任何健康卡号包括非数字字符, ALTER TABLE 将不能将数据类型从 CHARACTER 转换为 INTEGER。

技巧 42 使用 ALTER TABLE 语句改变主键和外键

除了改变列的宽度和数据类型之外, 还可使用 ALTER TABLE 添加诸如 PRIMARY KEY 和 FOREIGN KEY 一类的表约束。技巧 43 “使用 CREATE TABLE 语句指定主键”和技巧 44 “使用 CREATE TABLE 语句指定外键约束”将向读者展示在创建新的数据库表时如何作为 CREATE TABLE 语句的一部分定义键。对于已有的表来说, 可使用 ALTER TABLE 语句添加命名的和未命名的 PRIMARY KEY 和 FOREIGN KEY 约束。

PRIMARY KEY 和 FOREIGN KEY 约束两者都是数据库“键”, 这就意味着, 每一个都是惟一地标识表行的一列或几列的组合。虽然 PRIMARY 键惟一地标识了在其内定义了 PRIMARY 键的表中的一行, 但 FOREIGN KEY 却惟一地标识了另一表中的一行(一个表中的 FOREIGN KEY 总是引用另一表中的 PRIMARY 键)。

一个表只能有一个 PRIMARY KEY, 但它可有几个 FOREIGN KEYS。虽然组成 PRIMARY 键的列值和列的组合必须在表中对每一行来说是惟一的, 但组成 FOREIGN KEY 的列值或列的组合在定义 FOREIGN KEY 的表内不必是(大多数情况下也不是)惟一的。

读者将在技巧 129 “使用 PRIMARY KEY 列约束惟一地确定表行”中学习有关 PRIMARY KEY 约束的更多知识。

使用 ALTER TABLE 语句向表中添加 PRIMARY KEY 约束的句法如下:

```
ALTER TABLE <table name> ADD CONSTRAINT  
    <constraint name> PRIMARY KEY (<column name>  
    [, <column name>, ..., <last column name>])
```

因而, 如果由以下语句创建了一个表:

```
CREATE TABLE employee  
    (employee_id INTEGER NOT NULL,  
    badge_number SMALLINT NOT NULL,  
    first_name VARCHAR(20),  
    last_name VARCHAR(30))
```

可使用以下语句基于 EMPLOYEE_ID 添加单列 PRIMARY KEY:

```
ALTER TABLE employee  
ADD CONSTRAINT pk_employee PRIMARY KEY (employee_id)
```

如果单列中的值在表中的每一行中不是惟一的, 但列组合的值却是惟一的, 这时必须使用复合的或是多列的 PRIMARY KEY。例如, 假设有一个 EMPLOYEE 表, 其中两个雇员有相同的雇员号, 但没有两个具有相同雇员号的雇员又有相同的证章号。为了向表中添加 PRIMARY KEY, 可使用如下的 ALTER TABLE 将 EMPLOYEE_ID 和 BADGE_NUMBER 组合成单一的对表中的每一行都是惟一的键值:

```
ALTER TABLE employee  
ADD CONSTRAINT pk_employee  
PRIMARY KEY (employee_id, badge_number)
```

注意: 由于一个表只能有一个 PRIMARY KEY, 如果想要改变表的 PRIMARY KEY, 只要首先使用以下 ALTER TABLE 语句:

```
ALTER TABLE <table name> DROP CONSTRAINT <constraint name>
```

换一种说法就是，在可以使用 ALTER TABLE 语句向表中添加新的 PRIMARY KEY 约束之前，必须删除表中已有的 PRIMARY KEY。

向表中添加 FOREIGN KEY 约束与定义 PRIMARY KEY 类似。但是，当处理 FOREIGN KEY 时，不仅必须确定在当前表中的组成 FOREIGN KEY 的列，而且还要确定由 FOREIGN KEY 引用的表中的 PRIMARY KEY 列。

FOREIGN KEY 约束通常用于表示两个表间的父/子关系。当对一列或列的组合施加 FOREIGN KEY 约束时，就是说，在子记录（子表中的行）中的列（或列的组合）值可在父记录（父表中的行）的组成 PRIMARY KEY 的列（或列组合）中找到。

使用 ALTER TABLE 语句向表中添加 FOREIGN KEY 约束的句法如下：

```
ALTER TABLE <table name>
    [WITH NOCHECK]
    ADD [CONSTRAINT <constraint name>] FOREIGN KEY
        (<column name>[,<column name>...,<last column name>]
    REFERENCES <foreign table name>
        (<foreign column name>
        [,<foreign column name>...,<last foreign column name>])
```

由此，如果有一个用以下语句创建的父亲表 CUSTOMER：

```
CREATE TABLE customer
    (customer_number INTEGER PRIMARY KEY,
    first_name VARCHAR(20),
    last_name VARCHAR(30),
    address VARCHAR(35))
```

和由以下语句创建的子表 ORDER：

```
CREATE TABLE order
    (placed_by_customer_num INTEGER,
    order_date DATETIME,
    item_number INTEGER,
    quantity SMALLINT)
```

为了表示这两个表的父子关系，可使用以下 ALTER TABLE 语句：

```
ALTER TABLE order ADD
    CONSTRAINT fk_order_column
    FOREIGN KEY (placed_by_customer_num)
    REFERENCES customer (customer_number)
```

这就将 ORDER 表中的每一行与 CUSTOMER 表中的--行（而且只有一行）链接起来。换句话说，ORDER 表中的每一 PLACED_BY_CUSTOMER_NUM 列中的值可在 CUSTOMER 表的 CUSTOMER_NUMBER 列中找到。这样一来，每个定单（子）必须与发出该定单的顾客（父）相关联。

注意：当使用 ALTER TABLE 语句向表中添加时，DBMS 将检查已有的数据，确保其不会侵犯约束。在本例中，DBMS 将保证 ORDER 表中的 PLACED_BY_CUSTOMER_NUM 列的每个值都存在于 CUSTOMER 表的 CUSTOMER_NUMBER 列中。如果检查失败，DBMS 将不会创建 FOREIGN KEY，从而维护了引用数据的完整性。

如果能够确信已有的数据不会侵犯 FOREIGN KEY 约束,就可通过添加 WITH NOCHECK 子句来加速 ALTER TABLE 语句的执行。如果这样做了, DBMS 将不向表中已有行施加 FOREIGN KEY 约束。只有那些后来更新的或是插入的行才进行检查,以确保 FOREIGN KEY 值存在于被引用(父)表中的 PRIMARY KEY 列。

技巧 43 使用 CREATE TABLE 语句指定主键

键(key)惟一地标识表中的一行的列或列的组合。因而,键给出了一种在表中将一行与其他所有行区别开来的方法。由于一个键必须是惟一的,所以不应该在组成键的任何列中包括 NULL 值。请记住, DBMS 不对列中实际的 NULL 值做什么假设。这样一来,带有键列为 NULL 的一行将不能与表中其他行相区分,因为 NULL 值事实上可能等于其他行中的值。

每个表可有一个(而且只有一个) PRIMARY KEY。由于 PRIMARY KEY 必须是惟一地标识表中的每一行, DBMS 自动地向组成 PRIMARY KEY 的每一列施加 NOT NULL 约束。当创建新表时,可通过在列定义中包括关键词 PRIMARY KEY 创建单列的 PRIMARY KEY。

在 CREATE TABLE 语句中的 PRIMARY KEY 定义的句法有两种形式。如果 PRIMARY KEY 是作为 PRIMARY KEY 列定义的一部分定义的未命名的约束,则为:

```
[CONSTRAINT <constraint name>] PRIMARY KEY
```

而对于多列(复合的) PRIMARY KEY 或是命名的单列 PRIMARY KEY,其形式为:

```
CONSTRAINT <constraint name>  
PRIMARY KEY (<column name>  
[, <column_name>...[, <last column name>]])
```

例如,以下 CREATE TABLE 语句将 EMPLOYEE_ID 列标识为 EMPLOYEE 记录中的 PRIMARY KEY:

```
CREATE TABLE employee  
(employee_id INTEGER PRIMARY KEY,  
first_name VARCHAR(20),  
last_name VARCHAR(30))
```

因此, EMPLOYEE 表中的每一行必须在 EMPLOYEE_ID 列有惟一的非 NULL 的值。

系统在系统表中将 PRIMARY KEY 保存为约束。因而,如果没有为 PRIMARY KEY 起一个名字, DBMS 将为用户生成一个。如果想要删除表的 PRIMARY KEY,从而可使其变化,系统赋予的名称对用户来说变得很重要(因为一个表只能有一个 PRIMARY KEY,当想要改变 PRIMARY KEY 的列时,必须先将已有的 PRIMARY KEY 删除,然后再创建一个新的)。

注意:如果正在使用 MS-SQL Server,可通过执行以下命令调用存储过程 sp_help 显示 DBMS 赋予 PRIMARY KEY 的名称:

```
EXEC sp_help <table name>
```

(其中 <table name> 是带有想了解其名称的 PRIMARY KEY 的表名)。

MS-SQL Server 将用由 <table name> 标识的表的描述来响应该命令。可查看标题为 PRIMARY 的报告部分的 index_name 列,从中可查出 DBMS 赋予 PRIMARY KEY 的名称。

如果不想让 DBMS 生成其自己的 PRIMARY KEY 名称,也可在标识组成 PRIMARY KEY 的列时为其起名。例如,可使用以下 CREATE TABLE 语句,将名称 pk_employee_table 给予本例中的 PRIMARY KEY:

```
CREATE TABLE employee
  (employee_id INTEGER,
   CONSTRAINT pk_employee_table PRIMARY KEY,
   first_name VARCHAR(20),
   last_name VARCHAR(30))
```

有时表中没有单一列在每一行上都具有惟一值。例如，假设公司的每个分部发出其自己的雇员号。分部#1 有雇员#123、#124 和#126，分部#2 有雇员#121、#122 和#123。如果将 EMPLOYEE_ID 标识为 PRIMARY KEY，就可将分部#1 中的所有雇员插入 EMPLOYEE 表中。但当试图插入分部#2 的雇员#123 时，DBMS 将不允许这样做。由于 EMPLOYEE 表已经有了一行其 EMPLOYEE_ID 列的值为 123，DBMS 拒绝添加在 EMPLOYEE_ID 列中有值为 123 的第二个行，因为 PRIMARY KEY 为 EMPLOYEE_ID 列，必须在表的每一行中都是惟一的。

但仍然可为那种没有单列是惟一的表创建 PRIMARY KEY，只要将几个列标识（当一起看时，在表中的每一行上都是不同的）为 PRIMARY KEY。在本例中，读者知道雇员号对于分部来说是惟一的。因而，可使用由 EMPLOYEE_ID 和 DIVISION 组成的两列 PRIMARY KEY，例如，以下语句为 EMPLOYEE 创建 PRIMARY KEY：

```
CREATE TABLE employee
  (employee_id INTEGER,
   division SMALLINT,
   first_name VARCHAR(20),
   last_name VARCHAR(30)
  CONSTRAINT pk_employee_table
   PRIMARY KEY (employee_id, division))
```

注意：在 CREATE TABLE 语句内放置 PRIMARY KEY 列定义是不重要的。因而以下 CREATE TABLE 语句与本“注意”前面的例子是等价的：

```
CREATE TABLE employee
  (employee_id INTEGER
  CONSTRAINT pk_employee_table
   PRIMARY KEY (employee_id, division),
   division SMALLINT,
   first_name VARCHAR(20),
   last_name VARCHAR(30))
```

技巧 44 使用 CREATE TABLE 语句指定外键约束

正如前面所讨论的，数据库键唯一地标识了表中的一行。在技巧 43“使用 CREATE TABLE 语句指定主键”中，读者学习了 PRIMARY KEY 中的每一行唯一地标识了声明 PRIMARY KEY 的表内的单行。另一方面，FOREIGN KEY 引用与声明 FOREIGN KEY 的表不同的表内的 PRIMARY KEY。因而，一个表中的 FOREIGN KEY 内的每一行唯一地标识了另一表中的单行。虽然每个 PRIMARY KEY 值在表内必须是惟一的，但 FOREIGN KEY 内的值不需要是惟一的（而且很可能是不惟一的）。

FOREIGN KEY 通常代表两个表间的父子关系。当对一列（或列的组合）施加 FOREIGN KEY 约束时，可以说，在子表中的某行内的一列（或列的组合）的值可在父表特定行的组成 PRIMARY KEY 值的列（或列组合）中找到。

FOREIGN KEY 约束声明的句法如下:

```
[CONSTRAINT <constraint name>]
FOREIGN KEY (<column name>
    [,<column name>...[,<last column name>]])
REFERENCES <foreign table name>
    (<foreign table column name>
    [,<foreign table column name>...
    [,<last foreign table column name>]])
```

例如, 假设想要 CUSTOMER (父) 表和 ORDER (子) 表跟踪顾客定单, 这两个表由以下语句创建:

```
CREATE TABLE customer
    (customer_number INTEGER,
    first_name VARCHAR(20),
    last_name VARCHAR(30),
    address VARCHAR(35),
    CONSTRAINT pk_customer_table
    PRIMARY KEY (customer_number))
CREATE TABLE order
    (placed_by_customer_num INTEGER
    FOREIGN KEY (placed_by_customer_num) REFERENCES
    customer(customer_number),
    order_date DATETIME,
    item_number INTEGER,
    quantity SMALLINT,

    CONSTRAINT pk_order_table
    PRIMARY KEY
    (placed_by_customer_num, order_date, item_number))
```

在 ORDER 表中定义的 FOREIGN KEY 告诉人们可在 CUSTOMER (父) 表中一行的 CUSTOMER_NUMBER 列的 PRIMARY KEY 中 (子表 ORDER 中) 的 PLACED_BY_CUSTOMER_NUM 列中找到。

由于列 PLACED_BY_CUST_NUM 在 ORDER 表中是“外部的”而不是“主”键, 因而可不止一个 ORDER 行带有相同的 PLACED_BY_CUST_NUM 列值, 这就表明, 单独的顾客定了不止一种货品或是发出了不止一份定单。

FOREIGN KEY 对 PLACED_BY_CUSTOMER_NUM 列的约束还告诉人们, PLACED_BY_CUST_NUM 列中的值将在 CUSTOMER 表的 CUSTOMER_NUMBER 字段上出现一次, 而且只出现一次 (因为表中的 FOREIGN KEY 总是引用另一表中的 PRIMARY KEY)。因而, 人们将能够唯一地标识发出定单的顾客, 因为在当前表中的 FOREIGN KEY 值 (PLACED_BY_CUSTOMER_NUM) 唯一地标识了“外部” (CUSTOMER) 表中的一行。

当未为 FOREIGN KEY 约束提供名称时, 系统将为用户生成一个, 这样才能将约束保存在系统表中。在本例中, ORDER 表中的 FOREIGN KEY 没有明确地命名。因而系统将生成一个名字。

如果正在使用 MS-SQL Server, 可通过执行存储过程 sp_help 并为以下语句中的 <table name> 部分提供父表 (在本例中是 CUSTOMER) 的名称, 从而确定 FOREIGN KEY 的名称:

```
EXEC sp_help <table name>
```

MS-SQL Server 将以表的描述来响应上述命令而且在报告的 Table Is Referenced By (被……引用的表) 部分列出了 FOREIGN KEY 的名称。

如果想要为 FOREIGN KEY 选择一个名字 (而不是让 DBMS 为其生成名字), 可将本例中的 CREATE TABLE 语句改变如下:

```
CREATE TABLE order
(placed_by_customer_num INTEGER
  CONSTRAINT fk_customer_table FOREIGN KEY
    (placed_by_customer_num) REFERENCES
      customer(customer_number),
  order_date DATETIME,
  item_number INTEGER,
  quantity SMALLINT,
  CONSTRAINT pk_order_table
    PRIMARY KEY
      (placed_by_customer_num, order_date, item_number))
```

正如在技巧 43 中所学到的, PRIMARY KEY 可由不止一列组成。当一个表中的 FOREIGN KEY 引用另一表中的复合的 (或多列的) PRIMARY KEY 时, FOREIGN KEY 也将由多列组成。例如, 假设本例中定单的 ITEM_MASTER 表由以下语句所定义:

```
CREATE TABLE item_master
(item_number INTEGER,
  vendor_id INTEGER,
  quantity_on_hand SMALLINT,
  CONSTRAINT pk_item_master_table
    PRIMARY KEY (item_number, vendor_id))
```

使用以下 CREATE TABLE 语句, 可以在 ITEM_MASTER 表中使用 FOREIGN KEY 约束 (FK_ITEM_MASTER_TABLE) 引用复合的 PRIMARY KEY:

```
CREATE TABLE order
(placed_by_customer_num INTEGER
  order_date DATETIME,
  item_number INTEGER,
  vendor_id_number INTEGER,
  quantity SMALLINT,
  CONSTRAINT fk_item_master_table FOREIGN KEY
    (item_number, vendor_id_number) REFERENCES
      item_master (item_number, vendor_id),
  CONSTRAINT fk_customer_table FOREIGN KEY
    (placed_by_customer_num) REFERENCES
      customer(customer_number),
  CONSTRAINT pk_order_table
    PRIMARY KEY
      (placed_by_customer_num, order_date, item_number))
```

技巧 45 使用 MS-SQL Server Enterprise Manager Create View Wizard 创建视图

视图是虚拟表。虽然其外观和行为像是通常的关系数据库表, 但视图不包括数据。实际

上视图是一套 DBMS 告诉它保存在物理 (实际) 表中的什么数据要显示以及如何显示的指令。MS-SQL Server 给出两种定义视图的方法。可以使用 CREATE VIEW 语句 (读者将在技巧 153 “使用视图显示一个或多个表或视图中的列” 中学习有关知识), 或是使用 MS-SQL Server 的 Create View Wizard (创建视图向导)。不管使用哪种方法来创建视图, DBMS 将在系统表中与列出视图列的 SELECT 语句以及搜索标准 (其 WHERE 子句) 一起保存视图名。在最低的水平上, 所有视图都基于一个或多个物理数据库表 (可根据其他视图创建视图。但在某种程度上, 视图链中的视图之一一定要基于实际的数据库表)。因而, 为了看出如何使用 Create View Wizard 来创建视图, 开始必须决定想要显示的数据。例如, 假设想要根据如图 2.9 中所示的 PRODUCTION 表中的数据来创建视图。

PRODUCTION (销售情况) 表

Rep_ID	Call	Appointments	Sales	Deliveries
1	100	4	3	2
2	255	7	4	4
3	750	12	6	5
4	400	15	9	7
5	625	10	8	6
6	384	11	6	4
7	295	17	4	1

图 2.9 带有用于创建视图的示例数据的 PRODUCTION 表

为了使用 Create View Wizard 来创建显示从单个表中来的数据的视图, 可执行以下步骤:

1. 为了启动 Enterprise Manager, 可单击 Start 按钮, 将鼠标指针移动到 Start 菜单的 Programs 上并选择 Microsoft SQL Server 7.0, 然后单击 Enterprise Manager。
2. 为了显示 SQL 服务器的列表, 可单击 SQL Server Group 左边的加号 (+)。
3. 单击包括想要创建视图的数据库的 SQL Server 图标。例如, 如果想要在名为 NVBizNet2 的 SQL Server 上的数据库中创建视图, 可单击代表 NVBizNet2 的图标。
4. 选择 Tools 菜单上的 Wizards 选项 (或单击 Wizards 按钮, 即标准工具栏上的魔术棒)。Enterprise Manager 将显示 Select Wizard (选择向导) 对话框, 在其上可选择想要使用的向导。
5. 单击 Database 左边的加号 (+), 显示数据库对象向导的列表。
6. 单击 Create View Wizard (创建视图向导) 将其选择, 然后单击 OK 按钮。Enterprise Manager 将启动 Create View Wizard, 显示其 Welcome to the Create View Wizard (欢迎来到创建视图向导) 屏幕。
7. 单击 Next 按钮。Create View Wizard 将显示 Select Database (选择数据库) 对话框。
8. 单击 Database Name 名称区域右边的下拉列表按钮, 显示在步骤 3 中所选的 SQL Server 上的数据库的清单。
9. 单击想要在其上创建视图的数据库。对本例来说, 单击选择 SQLTips 数据库。
10. 单击 Next 按钮。Create View Wizard 向导将显示 Select Tables (选择表) 对话框, 如图 2.10 所示。



图 2.10 MS-SQL Server Create View Wizard 的 Select Tables 对话框

11. 单击想要将其数据包括在视图中的表的复选框。对本例来说，单击 PRODUCTION 表的复选框，使之出现选择标记。

12. 单击 Next 按钮。Create View Wizard 将显示 Select Columns（选择列）对话框，如图 2.11 所示。

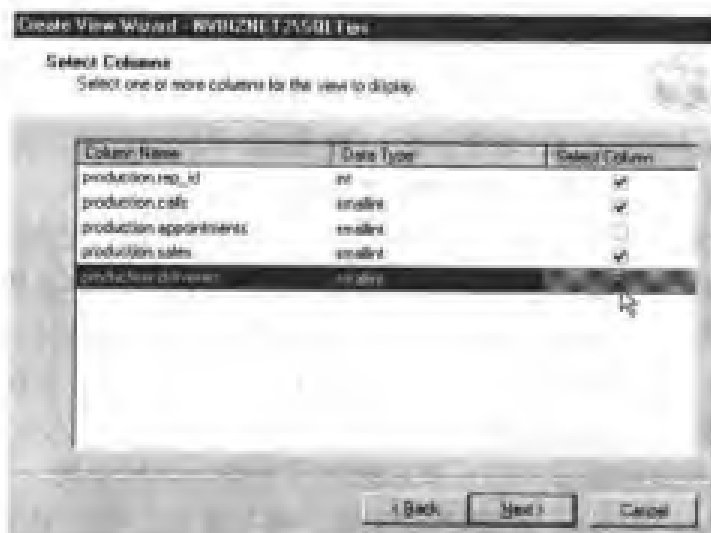


图 2.11 MS-SQL Server Create View Wizard 的 Select Columns 对话框

13. 单击想要显示的列的复选框。对本例来说，选择 PRODUCTION.REP_ID、PRODUCTION.CALLS、PRODUCTION.SALES 和 PRODUCTION.DELIVERIES 列。

注意：在 Select Columns 对话框的选择区域中的列的清单包括步骤 11 中所选的所有表中的所有列。Create View Wizard 使用合格的列名来显示哪个列属于哪个表，也就是说，它将列名显示为 <table name>.<column name>（其中 <table name>是包括 <column name>列的表名）。

14. 单击 Next 按钮。Create View Wizard 将显示 Define Restriction（定义限制）对话框。如果不想显示在步骤 11 中所选表中的所有行，可键入 WHERE 子句，让 DBMS 作为选择要显

示的行的标准。对本例来说, 键入 `WHERE PRODUCTION.SALES > 4` 让 DBMS 只显示 `PRODUCTION` 表中 `SALES` 列中的值大于 4 的行。

15. 单击 Next 按钮。Create View Wizard 将显示 Name the View (为视图命名) 对话框。

16. 在 View Name 区域键入为视图起的名称, 对本例来说, 在 View Name 区域键入 `vw_sales_production`。

注意: 最好保持视图名的一致性, 这样在查看数据库对象的清单时, 可将其与实际表区分出来。例如, 如果所有视图名 (而且只有视图名) 都以 `vw_` 开头, 这时只要看到以 `vw_` 开头的数据库对象, 那肯定是视图。

17. 单击 Next 按钮。Create View Wizard 将在如图 2.12 所示的 Completing the Create View Wizard (结束创建视图向导) 对话框中显示 DBMS 用于创建视图的 SQL 语句。

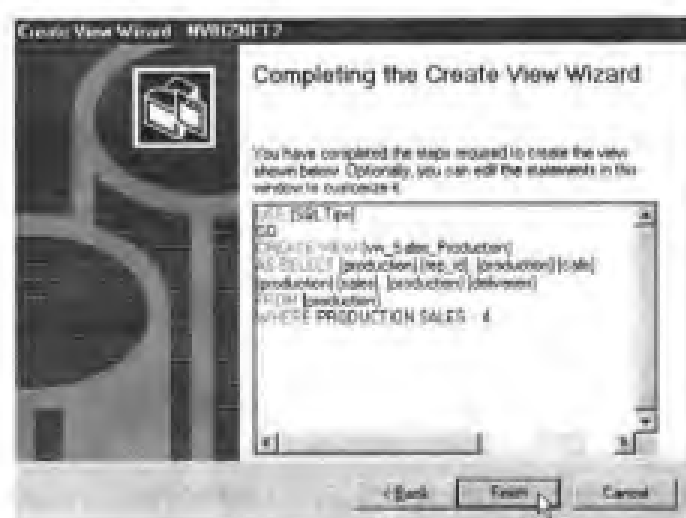


图 2.12 MS-SQL Server Create View Wizard 的 Completing the Create View Wizard 对话框

18. 对 `WHERE` 子句和其他必要的改变做些必要的修改, 以便进一步地改善视图 (可添加或删除列, 改变视图名, 改变 `WHERE` 子句中的选择标准等)。

19. 单击 Finish 按钮。

在完成步骤 19 之后, Create View Wizard 将检查 Completing the Create View Wizard (结束创建视图向导) 对话框中的语句的句法并提示改正任何错误 (如果有什么错误的话, 问题很可能出现在步骤 14 中键入的 `WHERE` 子句中)。

重复步骤 18 和步骤 19, 直到 Create View Wizard 显示 “Wizard Complete!” 消息框时为止, 单击 OK 按钮返回 MS-SQL Server Enterprise Manager 应用程序窗口。

一旦 DBMS 将视图保存到其系统表中, 可使用 `SELECT` 语句显示视图数据。例如, 为了显示在本例中所创建的视图中的所有行和列, 可执行以下 `SELECT` 语句:

```
SELECT * FROM vw_sales_production
```

假如 `PRODUCTION` 表具有如图 2.9 中所显示的数据, 则 MS-SQL Server 将把虚拟视图表 `VW_SALES_PRODUCTION` 的内容显示为:

rep_id	calls	sales	deliveries
3	750	6	5
4	400	9	7

```

5          625      8      6
6          384      6      4
(4 row(s) affected)

```

技巧 46 理解 DROP VIEW 语句中的 CASCADE 和 RESTRICT 子句

使用 DROP VIEW 语句删除视图时，DBMS 不会删除任何数据库数据。但是，仍然必须小心，看一下要删除的视图是否被其他视图引用。某些 DBMS 产品允许向 DROP VIEW 语句中添加 CASCADE 或 RESTRICT 子句，以便控制在删除被其他视图引用的视图时的 DBMS 的行为。

如果执行带 CASCADE 子句的 DROP VIEW 语句，DBMS 将不仅删除在 DROP VIEW 语句中指定的视图，而且还删除任何其他引用了 DROP VIEW 语句中的视图的视图。例如，如果有两个定义如下的视图：

```

CREATE VIEW vw_sales_production AS
SELECT rep_id, calls, sales, deliveries FROM production

```

和

```

CREATE VIEW vw_delivered_sales_commissions
(rep_id, deliveries, commission) AS
SELECT rep_id, deliveries, deliveries * 150.00
FROM vw_sales_production

```

当执行以下语句时：

```
DROP VIEW vw_sales_production
```

DBMS 只从系统表中删除 VW_SALES_PRODUCTION 视图。如果执行以下 SELECT 语句：

```
SELECT * FROM vw_delivered_sales_commissions
```

在删除 VW_SALES_PRODUCTION 视图之后，DBMS 用以下形式响应错误：

```

Server: Msg 208, Level 16, State 1,
Procedure vw_delivered_sales_commissions, Line 1
Invalid object name 'vw_sales_production'.
Server: Msg 4413, Level 16, State 1, Line 1
Could not use view 'vw_delivered_sales_commissions'
previous binding errors.

```

另一方面，如果使用以下 DROP VIEW 语句删除 VW_SALES_PRODUCTION 视图：

```
DROP VIEW vw_sales_production CASCADE
```

DBMS 从系统表中既删除 VW_SALES_PRODUCTION 也删除 VW_DELIVERED_SALES_COMMISSIONS 视图（后者引用了前者）。

作为对照，某些 DBMS 允许向 DROP VIEW 语句添加 RESTRICT 子句。RESTRICT 子句可防止删除被别的视图引用的视图。这样一来，在本例中，执行以下“受限制的” DROP VIEW 语句：

```
DROP VIEW vw_sales_production RESTRICT
```

将不会成功，因为 VW_SALES_PRODUCTION 视图被 VW_DELIVERED_SALES_COMMISSIONS 视图所引用。

注意：并非所有的 DBMS 产品都为 DROP VIEW 语句提供 CASCADE 和 RESTRICT 子句——例如，MS-SQL Server 就未提供。由此，请检查一下自己的系统文档，看一下自己的 DBMS 产品是否可向 DROP VIEW 语句添加 CASCADE 和 RESTRICT 子句。

第3章 使用 SQL 的数据操纵语言 (DML)

在 SQL 表内插入并操作数据

技巧 47 使用 INSERT 语句向表中添加行

使用 INSERT 语句可一次向数据库表中添加一行数据。INSERT 语句的句法如下：

```
INSERT INTO <table name>
    [(<column name>...[,<last column name>])]
VALUES (<column value>...[,<last column value>])
```

这就告诉 DBMS 向由 <table name> 给定的表中将值（在 VALUES 子句中列出的）插入列（在 <table name> 之后的 INTO 子句中列出的）。

虽然词汇 INSERT 好像是意味着在表中的已有的表行之前或已有的两行之间插入新的数据行，但 INSERT 语句只是建立一个新的与表的列结构相匹配的数据行。在接受了 INSERT 语句之后，DBMS 决定新行在表中的物理位置。新的数据行有可能出现在表的开头，也可能出现在已有的表行之间。

注意：正如读者在技巧 4 “理解 Codd 的 12 条关系数据库定义规则”中所学到的，关系数据库必须表现逻辑数据独立性（规则 9）。由此，行中实际列的顺序（只要在整个表中是一致的）以及表中的行顺序对查询数据库时提取的信息没有影响。因而，DBMS 将新的数据行放在表中的何处并不重要。

例如，假设有一个用以下语句创建的 ORDERS 表：

```
CREATE orders
    (order_number      INTEGER NOT NULL,
     customer_number   INTEGER NOT NULL,
     item_number       INTEGER NOT NULL,
     quantity         SMALLINT DEFAULT 1,
     order_date        DATETIME,
     special_instructions VARCHAR(30))
```

可使用以下 INSERT 语句添加由顾客 10 于 05/18/00 发出的、购买 5 件货品号为 1001 的、定单号为 1 的，而且对 ORDERS 表必须保持“冻结”状态的定单：

```
INSERT INTO orders
    (order_number, customer_number, item_number, quantity,
     order_date, special_instructions)
VALUES (1, 10, 1001, 5, '05/18/00', 'keep frozen')
```

在关键词 INSERT 之后列出的列名必须存在于 DBMS 要向其中插入行的表的表定义中。名称不必与表中的列的顺序相同，也不必列出所有的表列（正如读者将在技巧 50“使用 INSERT 语句向行的特定列中添加数据”中将要学习的）。因此，以下 INSERT 语句与前一段中给出的是等价的，虽然列名及其值是以不同顺序列出的：

```
INSERT INTO orders
(customer_number, order_number, order_date, item_number,
quantity, special_instructions)
VALUES (10, 1, '05/18/00', 1001, 5, 'keep frozen')
```

从左开始向右，DBMS 从 VALUES 子句中取出第一个值并将其放入列名清单中的第一个名称给定的表列中，然后从值列表中取出第二个值，将其放入由列名清单中的第二个名称指定的表列中，如此重复下去。列的清单和值的清单两者必须包括相等数目的数据项。另外，值清单中的每个值的数据类型必须与对应的表列的数据类型相兼容。

注意：大多数 DBMS 产品都允许从 INSERT 语句中忽略列清单。如果未提供列清单，则 DBMS 假设列清单中包括表中所有的列，且以在表定义中出现的顺序出现。因此，在当前的例子中

```
INSERT INTO orders
VALUES (1, 10, 1001, 5, '05/18/00', 'keep frozen')
```

等价于以下语句：

```
INSERT INTO orders
(order_number, customer_number, item_number, quantity,
order_date, special_instructions)
VALUES (1, 10, 1001, 5, '05/18/00', 'keep frozen')
```

为了得到表列的清单和其中列出现在表中每行的顺序，但不列出任何表数据，可执行以下的 SELECT 语句：

```
SELECT * <table name> WHERE NULL = NULL
```

（当然要用想要显示列的表名来代替其中的 <table name> 部分。）

技巧 48 使用 INSERT 语句通过视图插入行

除了使用视图显示表数据之外，还可在 INSERT 语句中使用视图向视图的底层表中添加数据。通过视图向表中插入数据的句法与直接向表中插入数据的句法是一样的，但要使用视图名而不是表名作为 INSERT 语句中 INSERT 子句的目标。这样一来，通过视图向表中添加行的句法如下：

```
INSERT INTO <view_name>
[(<view column name>...[,<view column name>]]
VALUES (<view column value>...[,<last value column value>])
```

例如，假若有一个表是由以下 SQL 语句创建的：

```
CREATE orders
(order_id    INTEGER IDENTITY,
cust_id     INTEGER NOT NULL,
item        INTEGER NOT NULL,
qty         SMALLINT DEFAULT 1,
order_date  CHAR(10),
ship_date   CHAR(10),
handling    VARCHAR(30) DEFAULT 'none')
```

而且视图的定义为：

```
CREATE VIEW vw_shipped_orders AS
SELECT order_id, cust_id, item, qty, order_date,
```

```

        ship_date
FROM orders
WHERE ship_date IS NOT NULL

```

可使用以下 INSERT 语句向底层表 ORDERS 中添加行:

```

INSERT INTO vw_shipped_orders
    (cust_id, item, qty, order_date, ship_date)
VALUES (1002, 55, 10, '2000-05-17', '2000-18-00')

```

对视图执行以下的 SELECT 语句:

```
SELECT * FROM vw_shipped_orders
```

将生成以下结果:

```

order_id  cust_id  item  qty  order_date  ship_date
-----
1         1002    55   10   2000-05-17  200-15-18
(1 row(s) affected)

```

对底层的 ORDERS 表执行以下 SELECT 语句:

```
SELECT * FROM orders
```

将生成以下结果:

```

order_id  cust_id  item  qty  order_date  ship_date  handling
-----
1         1002    55   10   2000-05-17  200-15-18  none
(1 row(s) affected)

```

请注意, 当通过视图插入数据时, 不用包括视图中的所有列中的值。在本例中, 包括在视图定义中的 ORDER_ID 列已经从 INSERT 语句中忽略掉。

正如直接向底层表中插入数据的情况那样, DBMS 将对未指定值的任何表列使用表列的默认值。在本例中, DBMS 将为 ORDER_ID 列提供“下一个”惟一的整数值, 因为表定义为 ORDER_ID 列指定了 IDENTITY 属性, 而且 INSERT 语句未为该列包括值 (在技巧 17 “理解 MS-SQL Server 的 IDENTITY 属性”中学习了有关知识)。类似地, DBMS 将使用为 HANDLING 定义的默认值“none”, 因为 INSERT 语句的值清单中未包括 HANDLING 列值。

注意: 如果列没有默认值而且 INSERT 语句未为其包括一个明确的值, DBMS 在执行 INSERT 语句时, 将把该列设置为 NULL。

注意: 当通过视图向底层表中插入行时, 只能为视图定义中的列提供值。在本例中, HANDLING 并不是视图定义的一部分。由此, 当通过 VW_SHIPPED_ORDERS 视图向 ORDERS 表中添加行时, 不能为 HANDLING 列提供值。

当通过视图向表中插入数据时, 可能会出现一个有趣的异常情况——当使用视图显示表数据时, 成功地执行了 INSERT 语句的一行似乎可能“消失”了。例如, 执行了以下 INSERT 语句通过 VW_SHIPPED_ORDERS 视图向 ORDERS 表中添加一行:

```

INSERT INTO vw_shipped_orders
    (cust_id, item, qty, order_date)
VALUES (2004, 110, 20, '2000-05-01')

```

接着, 执行以下 SELECT 语句:

```
SELECT * FROM vw_shipped_orders
```

DBMS 将返回以下结果:

```
order_id cust_id item qty order_date ship_date
-----
1      1002     55  10   2000-05-17 200-15-18
(1 row(s) affected)
```

第二个定单跑到哪里去了呢？

如果对底层的 ORDERS 表执行以下 SELECT 语句：

```
SELECT * FROM orders
```

DBMS 将返回以下结果：

```
order_id cust_id item qty order_date ship_date handling
-----
1      1002     55  10   2000-05-17 200-15-18 none
2      2004     110  20   2000-05-01 NULL      none
(2 row(s) affected)
```

这样来看，第二个定单确实在表中！

对视图使用 SELECT 语句并不显示第二个定单，但对底层的 ORDERS 表却能显示第二个定单的原因是，第二个定单中 SHIP_DATE 列的值是 NULL。由于视图定义中的 WHERE 子句指定了视图只显示其中 SHIP_DATE 列不是 NULL 的行，所以视图不显示第二个定单（由于 INSERT 语句未为该列包括值，DBMS 将 SHIP_DATE 列设置为 NULL）。

技巧 49 使用 MS-SQL Server Enterprise Manager 定义或改变主键约束

正如读者在技巧 43“使用 CREATE TABLE 语句指定主键”中所学到的，PRIMARY KEY 唯一地标识了其中定义了 PRIMARY KEY 的表中的每一行。MS-SQL Server 给出向表添加关键字约束的 3 种方法：作为 CREATE TABLE 语句的一部分（正如在技巧 43 和技巧 44“使用 CREATE TABLE 语句指定外键约束”中所学到的），作为 ALTER TABLE 语句的一部分（正如在技巧 42“使用 ALTER TABLE 语句改变主键和外键”中所学到的），以及使用 MS-SQL Server Enterprise Manager。

为了使用 MS-SQL Server Enterprise Manager 来添加或改变表的 PRIMARY KEY 约束，可执行以下步骤：

1. 为了启动 Enterprise Manager，可单击 Start 按钮，将鼠标指针移动到 Start 菜单的 Programs 上并选择 Microsoft SQL Server 7.0，然后单击 Enterprise Manager。
2. 为了显示 SQL 服务器的列表，可单击 SQL Server Group 左边的加号 (+)。
3. 单击带有要在其中添加关键字约束的数据库的 SQL Server 左边的加号 (+)。例如，如果想要对名为 NVBizNet2 的 SQL Server 上的数据库中的表加以处理，可单击 NVBizNet2 左边的加号 (+)。Enterprise Manager 将显示包括在所选择的 MS-SQL Server 上的可用的数据库和服务的文件夹。
4. 单击 Databases 文件夹左边的加号 (+)。Enterprise Manager 将展开 SQL Server 的数据库分支以显示在步骤 3 中选择的 SQL Server 上的数据库列表。
5. 单击包括想在其中添加关键字约束的表的数据库左边的加号 (+)。对本例来说，单击 SQLTips 数据库图标左边的加号 (+)。Enterprise Manager 将显示代表数据库对象的图标清单。
6. 单击 Tables 图标。Enterprise Manager 将在应用程序的右窗格中显示数据库中的表清单。

(其对象清单已在步骤 5 中展开)。

7. 单击想要添加关键字约束的表名。对本例来说, 单击 EX_ORDERS。

8. 选择 Action 菜单中的 Design Table (设计表) 选项。Enterprise Manager 将显示 Design Table 窗口, 如图 3.1 所示。

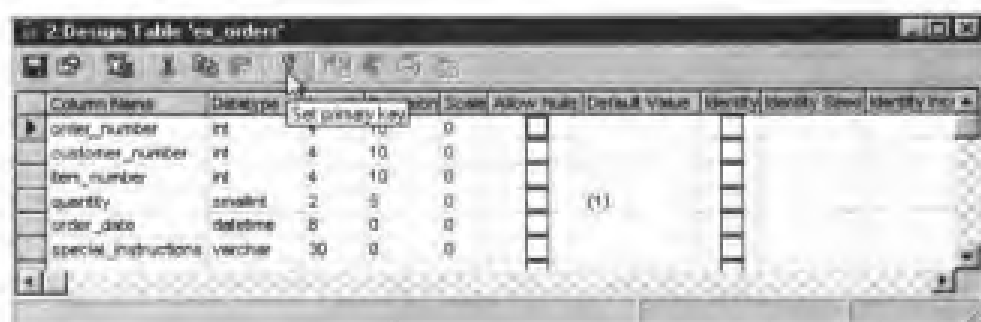


图 3.1 MS-SQL Server Enterprise Manager 的 Design Table 窗口

9. 为了为表的 PRIMARY KEY 选择单列, 可单击组成关键字的列名左边的选择按钮。如果想要创建复合的或多列的 PRIMARY KEY, 可在按下 Ctrl 键的同时单击要包括在键中的列名的选择按钮。对于本例来说, 单击 ORDER_NUMBER 左边的选择按钮。

注意: 一定要为 PRIMARY KEY 中的每一列清除 Allow Nulls 复选框。在组成 PRIMARY KEY 的列中, NULL 值是不允许的。

10. 为了告诉 Enterprise Manager 创建 PRIMARY KEY, 既可在 Design Table 窗口的标准工具栏上单击 Set Primary Key (设置主键) 按钮, 也可右击 PRIMARY KEY 中的列之一, 然后在弹出的菜单中选择 Set Primary Key 选项。

11. 为了保存改变, 单击 Save 按钮 (软盘图标), 即 Design Table 窗口的标准工具栏上左数的第一个按钮。

12. 单击 Design Table 窗口右上角的关闭按钮 (X)。

当完成步骤 12 时, Windows 将关闭 Design Table 窗口并返回 Enterprise Manager 应用程序窗口。

技巧 50 使用 INSERT 语句向行的特定列中添加数据

INSERT 语句的最常用的形式是作为行值表达式。虽然可使用 INSERT 语句向表中添加单个值, 但通常将其用于一次添加一行数据。例如, 下面的 INSERT 语句:

```
INSERT INTO employee VALUES (1, 'Konrad', 'King',
    '555-55-5555', '7810 Greenwood Ave', NULL, NULL)
```

将向使用以下语句创建的表中添加一行:

```
CREATE TABLE employee
(
    employee_id          INTEGER,
    first_name           VARCHAR(20),
    last_name            VARCHAR(30) NOT NULL,
    social_security_number CHAR(11),
    street_address       CHAR(30),
    health_card_number   CHAR(15),
```

```
sheriff_card_number    CHAR(15),
PRIMARY KEY (employee_id))
```

即将 VALUES 子名中的数据值按位置转移到表行的列中。“按位置”意味着，DBMS 将第一个值 1 放入第一列 `employee_id` 中，将第二个值 Konrad 放入第二列 `first_name` 中，第三个值 King 放入第三列 `last_name`，依次类推。

INSERT 语句的句法如下：

```
INSERT INTO <table name>
    [(<column name>[...,<last column name>])]
VALUES (<column value>[...,<last column value>])
```

如果在 <table name> 后面忽略了可选的列清单，则 DBMS 将列的清单设置为包括表中所有的列，其顺序为在表定义中出现的顺序。在本例中，DBMS 将以下语句：

```
INSERT INTO employee VALUES (1, 'Konrad', 'King',
    '555-55-5555', '7810 Greenwood Ave', NULL, NULL)
```

看作好像是以下语句：

```
INSERT INTO employee
    (employee_id, first_name, last_name,
    social_security_number, street_address,
    health_card_number, sheriff_card_number)
VALUES (1, 'Konrad', 'King', '555-55-5555',
    '7810 Greenwood Ave', NULL, NULL)
```

虽然 VALUES 子句中的数据值必须与列清单（跟在表名后的）中指定的列的数目相同，但不必在列清单中列出表行的所有列。例如，如果想要只为 `EMPLOYEE_ID`、`FIRST_NAME` 和 `LAST_NAME` 提供值，可使用以下 INSERT 语句：

```
INSERT INTO employee
    (employee_id, first_name, last_name)
VALUES (2, 'Sally', 'Fields')
```

DBMS 接着把 `EMPLOYEE_ID` 列设置为 2，把 `FIRST_NAME` 列设置为 Sally，把 `LAST_NAME` 列设置为 Fields，而且行中的其余列为其默认值（这些值为 NULL，除非通过在 CREATE 语句中的 DEFAULT 子句将其设置为另外的值或者是以前使用存储过程 `sp_bindefault` 绑定的默认值）。

这样一来，从本例中可以看出，虽然在 INSERT 语句的列名清单中的每一列必须与插入数据的表中的列名匹配，但 INSERT 语句的列清单不必包括所有表列的名称。另外，列清单中的列名不必按其在表行中的顺序列出。例如，以下 INSERT 语句：

```
INSERT INTO employee
    (first_name, employee_id, last_name)
VALUES ('Sally', 2, 'Fields')
```

与以下语句等价：

```
INSERT INTO employee
    (employee_id, first_name, last_name)
VALUES (2, 'Sally', 'Fields')
```

在 INSERT 语句中列出列名的好处是，用户向表中添加了一列之后，带有列清单的语句仍然可正确地执行。例如，如果用户执行了以下语句：

```
ALTER TABLE employee ADD COLUMN hourly_pay_rate NUMERIC
```

以下 INSERT 语句将能成功执行, 因为 VALUES 子句没有为 HOURLY_PAY_RATE 列提供值:

```
INSERT INTO employee VALUES (1, 'Konrad', 'King',  
    '555-55-5555', '7810 Greenwood Ave', NULL, NULL)
```

请记住, 如果不提供列清单, DBMS 将列清单设置为包括表定义中的所有列名。这样一来, 在本例中的 INSERT 语句由于列名清单比 VALUES 子句中的值的数目多包括了一个列名, 而不能成功执行。

作为对照, 在使用同样的 ALTER TABLE 语句 (在前段中引用的) 向表中添加新列时, 以下语句仍能正确执行:

```
INSERT INTO employee  
    (first_name, employee_id, last_name)  
VALUES ('Sally', 2, 'Fields')
```

此句告诉 DBMS 将值放入 FIRST_NAME、EMPLOYEE_ID 和 LAST_NAME 列, 而将其余列 (包括新插入的 HOURLY_PAY_RATE 列) 设置为列的默认值 (NULL, 除非另外定义)。

注意: 当在 INSERT 语句中列出列时, 必须用 NOT NULL 约束为每一列包括值, 除非列有非 NULL 的默认值。在本例中, INSERT 语句必须既为 EMPLOYEE_ID 列也为 LAST_NAME 包括值, 因为这两个列都是由 NOT NULL 支配的, 而且两者都没有非 NULL 默认值 (对 EMPLOYEE_ID 列的 PRIMARY KEY 约束默认地向该列添加了 NOT NULL 约束)。由此以下 INSERT 语句:

```
INSERT INTO employee  
    (first_name, last_name) VALUES ('Sally', 'Fields')
```

在 DBMS 试图将 EMPLOYEE_ID 列设为 NULL 时将不能成功执行。

注意: 如果想让 DBMS 为单列 PRIMARY KEY 提供惟一的非 NULL 值, 可向该列添加 IDENTITY 属性 (读者已经在技巧 17 “理解 MS-SQL Server 的 IDENTITY 属性” 中学习过有关 IDENTITY 属性的内容)。如果本例中的 EMPLOYEE 表中的 EMPLOYEE_ID 列有 IDENTITY 属性, 则 DBMS 通过提供惟一的、非 NULL 的 EMPLOYEE_ID 值, 会成功执行前面 “注意” 中的 INSERT 语句。

技巧 51 使用 INSERT 语句将一个表中的行插入另一表

虽然 INSERT 语句主要用于向表中添加一行数据, 但也可将单条 INSERT 语句用于一次向表中添加多行。事实上, 可使用 INSERT 语句将一个表的全部或部分复制到另一表中。

正如在技巧 50 “使用 INSERT 语句向行的特定列中添加数据” 中所学到的, INSERT 语句的句法是:

```
INSERT INTO <table name>  
    [(<column name> [...,<last column name>])]  
VALUES (<column value>[...,<last column value>])  
当然, INSERT 语句的 VALUES 子句应该实际上写为:  
INSERT INTO <table name>  
    [(<column name> [...,<last column name>])]  
VALUES (<row value constructor>
```

```
{..., <last row value constructor>}]
```

其中每个<row value constructor>是与语句中的列名清单（跟随在<table name>后）中的列数匹配的列值的清单。

因而，虽然在技巧 50 中使用 INSERT 语句向表中添加单行，但在想要向表中一次添加多行时，也可向同一语句添加额外的行值。例如，以下语句：

```
INSERT INTO employee VALUES (1, 'Konrad', 'King',
    '555-55-5555', '7810 Greenwood Ave', NULL, NULL)
```

带有一个行值构建器（由关键词 VALUES 引入的），将把一行添加到 employee 表中，而以下语句：

```
INSERT INTO employee
    (employee_id, first_name, last_name,
    social_security_number, street_address,
    health_card_number, sheriff_card_number)
VALUES (1, 'Konrad', 'King', '555-55-5555',
    '7810 Greenwood Ave', NULL, NULL),
    (2, 'Sally', 'Fields', '556-55-5555',
    '77 Sunset Strip', NULL, NULL),
    (3, 'Wally', 'Wallberg', '557-55-5555',
    '765 E. Eldorado Lane', NULL, NULL)
```

带有 3 个行值构建器（跟随于关键词 VALUES 之后），将向同一表中添加 3 行。

注意：正如以前所学过的，在前面段落中的两个 INSERT 语句的例子中，列清单（跟随于表名之后）是等价的。当未指定列名时（如在第一条 INSERT 语句中那样），DBMS 将为 INSERT 语句提供由表中所有列组成的列名清单（如在第二条 INSERT 语句中明确地枚举的那样）。

到目前为止，在本技巧中的 INSERT 语句列出了表行中的所有列的值。但是，正如单行 INSERT 语句的情况，多行的 INSERT 语句也可为行中的列提供全部或部分值。例如，以下语句：

```
INSERT INTO employee (employee_id, first_name, last_name)
VALUES (4, 'Joe', 'Kernan'), (5, 'David', 'Faber'),
    (3, 'Brad', 'Woodyard')
```

将向 EMPLOYEE 表添加 3 行（DBMS 将为未列在列名清单中的每一列提供列默认值[在本例中为 NULL]）。

由于 SQL 允许使用单条 INSERT 语句向表中一次添加多行，因而可用 SELECT 语句代替其中的 VALUES 子句，只要由 SELECT 语句构建的行具有与列名清单（在 INSERT 语句中跟随于表名之后）中同样数目和类型的列即可。

这样一来，为了将一个表中的行插入另一个表，可使用 INSERT 语句的如下句法：

```
INSERT INTO <table name>
    [(<column name> [...,<last column name>])]
SELECT <column name> [...,<last column name>]
FROM <table name>
[WHERE <search condition>]
```

因而，为了将 EMPLOYEE 表中所有雇员插入第二个雇员表 EMPLOYEE2 中，可使用以下 INSERT 语句：

```
INSERT INTO employee2 SELECT * from employee
```

注意：为了使示例中的 INSERT 语句中的 SELECT *子句有效，EMPLOYEE 和 EMPLOYEE2 两个表必须都有同样数目的列，且列的顺序也一样。在以下各段中，读者将学习如何通过明确地列出列名而不是使用在 SELECT *子句中隐含的列名清单，从而绕过这一限制。现在要了解的重要事情是，可从一个表中将数据复制到另一表中，既可通过列出想要复制数据的特定列，也可忽略列清单，在这种情况下，DBMS 认为用户想要从一个表中将所有的列值复制到另一表中。

如果有几个对多个表中的数据的查询，用户将发现，如果首先将多个表中的数据合并到一个临时表中，然后作为对新的总计表的单表查询执行 SQL 语句，则查询较为简单而且执行得更快。将数据收集到单个数据中可避免让 DBMS 多次搜索多个表，重复读取并删除不满足搜索条件的同样的数据。

例如，假设 CUSTOMERS、EMPLOYEES 和 ORDERS 表定义如下：

```
CREATE TABLE customers
(customer_id INTEGER, first_name VARCHAR(30),
last_name VARCHAR(30), address VARCHAR(35),
city VARCHAR (20), state CHAR(2),
zip_code INTEGER, phone_number CHAR(12),
salesperson INTEGER net_due_days SMALLINT,
credit_limit NUMERIC)

CREATE TABLE employees
(employee_id INTEGER, first_name VARCHAR(30),
last_name VARCHAR(30), address VARCHAR(35),
ssan CHAR(11), salary NUMERIC,
low_quota SMALLINT, medium_quota SMALLINT,
high_quota SMALLINT, sales_commission NUMERIC)

CREATE TABLE orders
(order_id INTEGER, order_date DATETIME,
item_number INTEGER, quantity SMALLINT,
customer_id INTEGER, salesman_id INTEGER)

CREATE TABLE products
(product_id SMALLINT, description VARCHAR(40),
quantity_on_hand SMALLINT, item_cost NUMERIC)
```

通过以下语句可将数据组合到单个表中：

```
CREATE TABLE temp_report_table
(customer_ID INTEGER,
cust_first_name VARCHAR(30),
cust_last_name VARCHAR(30),
salesman_ID INTEGER,
salesman_first_name VARCHAR(30),
salesman_last_name VARCHAR(30),
order_date DATETIME,
order_item_number SMALLINT,
order_item_quantity SMALLINT,
order_total NUMERIC,
order_item_desc VARCHAR(40))
```

使用以下的 INSERT 语句：

```
INSERT INTO temp_report_table
(customer_id, cust_first_name, cust_last_name,
salesman_id, salesman_first_name, salesman_last_name,
order_date, order_item_number, order_item_quantity,
order_total, order_item_desc)
SELECT
customers.customer_id, customer.first_name,
customers.last_name, employee_id, employees.first_name,
employees.last_name, order_date, item_number, quantity,
(quantity * item_cost), description
FROM orders, customers, products, employees
WHERE orders.customer_id = customers.customer_id
AND product_id = item_number
AND employee_id = salesman_id
```

在从 4 个表中将数据组合成一个表之后，可使用单表的 SELECT 语句来显示来自一个或多个表中的数据。例如以下的 SELECT 语句：

```
SELECT
cust_first_name, cust_last_name, salesman_first_name,
salesman_last_name, order_item_quantity, order_total,
order_item_desc
FROM temp_report_table WHERE order_item_number = 5
```

将显示卖出货品 5 的销售人员的姓名、购买该货品的顾客姓名和日期、数量、总计和购买 5 号货品的定单的描述。

还可使用简单的单表查询来显示最大定单的量以及卖出该货品的销售人员的姓名：

```
SELECT salesman_id, salesman_first_name,
salesman_last_name, order_total
FROM temp_report_table
WHERE order_total =
(SELECT MAX(order_total) FROM temp_report_table)
```

在单表查询的两个例子中，如果没有首先将所有所需的数据组合到单个表中的话，必须执行一条 SELECT 语句和多表的 JOIN 语句。

技巧 52 将 MS-SQL Server 的 SELECT INTO/BULKCOPY 数据库选项设置为 TRUE 以便加速从表到表的数据转移

SQL 关系 DBMS 模型的实力之一在于其执行事务处理的能力。通过将几个 SQL 语句组合在一起作为一个单个的事务处理，DBMS 可维护数据，即使当操作需要成功地执行几个 SQL 语句时也是如此。如果部分事务处理失败，DBMS 使用数据库的事务处理日志“备份出”部分执行的事务处理并向受影响的表中恢复处理前的数据值。当未提交的语句返回原状时，表看起来就好像事务处理中的 SQL 语句未执行时一样。

将部分执行的甚至是成功地完成但却有错误的事务处理返回原状的能力通常要比维护事务处理日志的开销更有意义——只要问一下数据库管理员（能够对被用户使用错误的搜索条件执行了 DELETE 语句而错误地删除的重要的（或许是不可替代的）数据行取消删除）就知道

其意义了。

虽然这是对正常处理的保护措施，但当将多个表中的大量数据合并为单个的冗余的临时表时（如在技巧 51 中所做的），维护事务处理日志仍然要有不必要的开销。毕竟，如果 SELECT 语句在执行的中途失败，还能在原始表中获得原始的数据值。由此，可 TRUNCATE（截短）临时表（即从中删除一些数据）并重新进行数据合并。

如果许多用户正在把大量的行复制到冗余的（而且经常是临时的）表中，可通过将数据库的 SELECT INTO/BULKCOPY 选项设置为 TRUE 来改善 DBMS 的整体性能（而且加快数据的转移）。将 SELECT INTO/BULKCOPY 选项设置为 TRUE 就告诉 DBMS 将表到表的数据转移看作是大量的插入。DBMS 在大量插入时在其事务处理日志中存储较少的信息，这就减少了维护事务处理日志所需的开销，因而可使 DBMS 使用额外的资源以较短的时间来完成表到表的转移。

注意：将 SELECT INTO/BULKCOPY 选项设置为 TRUE 具有数据库范围内的后果。虽然 SELECT INTO/BULKCOPY 被设置为 TRUE，但 DBMS 保存比全部事务处理日志信息（有关所有具有 SELECT INTO 子句的 INSERT 语句的信息）要少的信息。但是，缺少 undo 选项并不是一个问题，因为在这种情况下，原始的表中仍然有原来的数据。

为了减少在表对表的数据转移时写入事务处理日志的信息量，可使用以下句法执行 sp_dboption 存储过程：

```
sp_dboption <database name>, 'SELECT INTO/BULKCOPY', TRUE
```

由此，为了提高 SQLTips 数据库中表间的数据转移速度，可执行以下语句：

```
EXEC SP_DBOPTION SQLTips, 'SELECT INTO/BULKCOPY', TRUE
```

如果以后还想维护全部的事务处理日志以便全部更新（包括表对表的数据转移），可执行以下语句：

```
EXEC <database name>, 'SELECT INTO/BULKCOPY', FALSE
```

其中，<database name>是想要对包括 SELECT INTO 子句的 INSERT 语句禁用大量复制处理的数据库。

注意：如果正在从另一计算机系统、另一种 DBMS 或是从大型的顺序文件中装载表数据，可使用 MS-SQL Server 的 BULK INSERT 语句或 BCP 实用工具——两者都允许在单独表的基础上（这与数据库范围内的 SELECT INTO/BULKCOPY 数据库选项不同）设置事务处理日志和其他选项。例如 BCP 实用工具允许定义源数据的格式，而且装载数据要比重复的单行 INSERT 语句快得多（参见 MS-SQL Server 的“在线书”文档，从中可得到对 BCP 实用工具的帮助信息）。

技巧 53 使用 UPDATE 语句改变列值

可使用 UPDATE 语句来改变单行上的一列或多列的值，或是改变单个表中选定的一些行上的多个列值。当然，为了在 UPDATE 语句中修改指定表中的数据，必须有对表的 UPDATE 访问权（读者将在技巧 103 中学习有关 SQL 对象权限的知识）。

UPDATE 语句的句法如下：

```
UPDATE <table name | view name>
```

```
SET <column name> = <expression>
```

```
[..., <last column name> = <last expression>]
```

```
[WHERE <search condition>]
```

由此, 当#3 销售员 Joe Smith 被提升为区域主管时, 可使用以下 UPDATE 语句将 Joe Smith 的所有顾客重新分配给其新的#9 销售员 Sally Fields:

```
UPDATE customers SET salesperson = 9 WHERE salesperson = 3
```

也可以使用以下 UPDATE 语句将顾客 Konrad King 的信用限制设置为\$10,000, 而将支付期设置为净 120 天:

```
UPDATE customers  
SET    credit_limit = 10000, net_due_days = 120  
WHERE first_name = 'Konrad' AND last_name = 'King'
```

UPDATE 语句中的 WHERE 子句确定表 (由 <table name> 给定的) 中的列值要加以修改的一行或多行。SET 子句提供要将其赋予满足 WHERE 子句中搜索条件的行的列值的清单。简短地说, UPDATE 语句一次一行地处理整个表 (在本例中是顾客表) 并更新那些搜索条件为 TRUE 的行上的列值。作为对照, UPDATE 语句将那些搜索条件计算为 FALSE 或 NULL 的行中的列数据保持不变。

正如从 UPDATE 语句的句法所看到的, SET 子句包括列赋值表达式的清单。表列名可在赋值清单中作为赋值目标只能出现一次。另外, 表达式必须产生一个与要赋值的列的数据类型相兼容的值 (例如, 不能将一个字符串赋给 NUMERIC 数据类型的列)。另外, 表达式必须是根据当前正在更新的行的列值可计算出来的, 而且不能包括任何子查询或是列函数 (如 SUM、AVG、COUNT 等)。

注意: 未在表达式或是在 UPDATE 语句的 WHERE 子句中引用的列值保持为整个表中的任何更新以前有的值。这样一来, 以下 UPDATE 语句:

```
UPDATE employees  
SET low_quota = (low_quota * 2), medium_quota =  
    (low_quota * 4), high_quota = (medium_quota * 8)  
WHERE low_quota = 1 AND medium_quota = 2
```

将把 LOW_QUOTA 设置为 2, 把 MEDIUM_QUOTA 设置为 4 ($4 * 1$ 而不是 $4 * 2$), 而把 HIGH_QUOTA 设置为 16 ($8 * 2$ 而不是 $8 * 4$)。

一定要确保不要忽略 WHERE 子句, 除非想要更新表中的所有行。例如,

```
UPDATE employees SET low_quota = low_quota * 1.5
```

将对 EMPLOYEES 表中的所有雇员把 LOW_QUOTA 值增加 150%。

注意: 在执行一条新的 UPDATE 语句 (特别是带有复杂选择标准的) 之前, 应使用 UPDATE 语句的 WHERE 子句执行 SELECT COUNT(*) 语句。例如, 以下语句:

```
SELECT COUNT(*) FROM customers WHERE salesperson = 3
```

将给出当执行以下 UPDATE 语句时 DBMS 将修改的行数:

```
UPDATE customers SET salesperson = 9 WHERE salesperson = 3
```

通过看出 UPDATE 语句将改变的行数, 就可能捕获选择标准中的错误——如果对所期望要修改的行数有某种概念的话。

技巧 54 使用带条件子句的 UPDATE 语句同时改变多行中的值

如果想要更新表中所有行中的列, 可使用没有 WHERE 子句的 UPDATE 语句, 例如以下语句:


```
UPDATE employee SET YTD_fed_tax_withheld = 0.00,  
    YTD_FICA_Employer = 0.00, YTD_FICA_Employee = 0.00,  
    YTD_gross_pay = 0.00
```

将把 EMPLOYEE 表中所有行上的 YTD_FED_TAX_WITHHELD、YTD_FICA_EMPLOYER、YTD_FICA_EMPLOYEE 和 YTD_GROSS_PAY 列的值设置为 0.0。

为了只修改表中某些行中的列值，可向 UPDATE 语句添加 WHERE 子句。当 DBMS 执行具有 WHERE 子句的 UPDATE 语句时，它选择满足搜索条件的行，然后对那些行一次处理一行，更新由 UPDATE 语句的 SET 子句指定的列值。例如，以下 UPDATE 语句：

```
UPDATE employees SET low_quota = 1, medium_quota = 2,  
    high_quota = 4  
WHERE low_quota IS NULL
```

将把那些在执行 UPDATE 语句之前 LOW_QUOTA 为 NULL 的行的 3 列 LOW_QUOTA、MEDIUM_QUOTA 和 HIGH_QUOTA 分别设置为 1、2 和 4。

UPDATE 语句的 SET 子句中的列名必须是目标表（在关键词后的表名）中的列。另外，UPDATE 语句的 SET 子句中的表达式不能包括任何子查询或列函数，而且必须可计算成与要赋值的列相兼容的数据类型。

不用管 SET 子句中表达式的顺序。由于用在表达式（以及在 WHERE 子句）中的每一列的值被设置为更新以前的列值，由一个表达式对列值所做的改变对同一 UPDATE 语句中的其他表达式没有影响。

例如，假设正要对 SALES 列具有值为 10 且 LOW_QUOTA、MEDIUM_QUOTA 和 HIGH_QUOTA 列分别具有值为 1、2 和 4 的行执行以下 UPDATE 语句：

```
UPDATE employees  
SET department = ' Main Room', sales = 0,  
    low_quota = sales, medium_quota = low_quota + 5,  
    high_quota = medium_quota + 5  
WHERE sales > low_quota AND department = 'Training'
```

UPDATE 语句将把 DEPARTMENT 设置为 Main Room、把 SALES 设置为 0、将 LOW_QUOTA 设置为 10，把 MEDIUM_QUOTA 设置为 6 而把 HIGH_QUOTA 设置为 7。虽然它把 LOW_QUOTA 值从 1 改为 10，在把 MEDIUM_QUOTA 设置为 LOW_QUOTA + 5 之前，在整个 SET 子句的表达式中，UPDATE 语句使用所有列的更新前的值（包括 LOW_QUOTA 列）。因而 UPDATE 语句将 MEDIUM_QUOTA 设置为 1 + 5 或 6，将 HIGH_QUOTA 设置为 2 + 5 或 7——即使 LOW_QUOTA 和 MEDIUM 限额在 DBMS 计算表达式之前有新的较高的值。

如果在本例中想要根据 SALES 列的值来设置各限额[带 quota 的各列]的值，则应在每个表达式中使用 SALES。以下 UPDATE 语句：

```
UPDATE employees  
SET department = ' Main Room', sales = 0,  
    low_quota = sales, medium_quota = sales + 5,  
    high_quota = sales + 10  
WHERE sales > low_quota AND department = 'Training'
```

将产生所期望的限额值为 10、15 和 20。

技巧 55 在 UPDATE 语句中使用子查询同时改变多行中的值

技巧 53 “使用 UPDATE 语句改变列值”和技巧 54 “使用带条件子句的 UPDATE 语句同时改变多行中的值”中展示了使用 WHERE 子句的 UPDATE 语句的示例，其中 WHERE 子句可使用比较运算符（=、>、<、<>、IS）来确定一行的适用性。UPDATE 语句还允许使用 SELECT 语句的结果来指定想要在目标表中更新的行。

例如，假设想要把销售额小于平均数的雇员重新分配给培训部。可使用以下 UPDATE 语句：

```
UPDATE employees SET department = 'Training'
WHERE department <> 'Training'
AND sales < (SELECT AVG(sales)
             FROM employees
             WHERE department <> 'Training')
```

另一例子是，假设想要把负责 5 个以上雇员的主管改变为经理。可使用以下 UPDATE 语句：

```
UPDATE employees SET job_title = 'Manager'
WHERE job_title = 'Supervisor'
AND 5 < (SELECT COUNT (*)
        FROM employees WHERE reports_to = employee_id)
```

在本例中，不能忽略 job_title = 'Supervisor' 这个搜索条件，因为这样，负责 5 个以上雇员的副主管也会将其职位改变为经理。

在 WHERE 子句中的子查询可嵌套任何次数，也就是说，UPDATE 语句的 WHERE 子句中的 SELECT 语句中也可在其 WHERE 子句有 SELECT 语句，而且在 WHERE 子句中可有 SELECT 语句，可以继续下去。

SQL-89 不允许 UPDATE 语句的 WHERE 子句中的任何层次上的 SELECT 语句引用正在更新的表。SQL-92 取消了这一限制，在计算对目标表中的列引用时，就好像目标表中的所有列都未更新一样。由此，在第一个例子中的 SELECT 语句将使用对 EMPLOYEES 表的每一行的 SALES 列的相同的平均值，虽然平均是对未在培训部中的雇员做出的，因为低产值的雇员已被移动到培训部，而 DBMS 仍按其固有方式来计算表。

技巧 56 使用 UPDATE 语句根据另一表中的值改变表的值

虽然 UPDATE 语句只允许改变单个表（其名称紧随关键词 UPDATE 之后出现）中的列值，但在 UPDATE 语句的 WHERE 子句中可使用任何可用的表（或是表的组合）。由此，可根据其他表的列值来决定在目标表中更新哪一行。

例如，假设用户销售来自多个经销商处的汽车零件，而经销商之一 XYZ Corp 已经倒闭。此时可使用以下 UPDATE 语句改变 INVENTORY 表中所有来自 XYZ Corp 的零件的 REORDER_STATUS 列：

```
UPDATE INVENTORY SET reorder_status = 'Discontinued'
WHERE vendor_id IN (SELECT vendor_id FROM vendors
                   WHERE company_name = 'XYZ Corp')
```

当然，这个例子离得太远了一点，因为通常知道 VENDOR_ID 并将其用在 WHERE 子句代替子查询，因而如果 XYZ Corp 的 VENDOR_ID 是 5 的话，可编写如下的 UPDATE 语句：

```
UPDATE INVENTORY SET reorder_status = 'Discontinued'
```

```
WHERE vendor_id = 5
```

作为另一个例子，假设想要将每一个其顾客下了多于\$1,000,000 的定单的销售员标识为“Key Account Manager”。用以下语句定义表：

```
CREATE TABLE employees
(
  employee_id INTEGER,
  first_name   VARCHAR(25),
  last_name    VARCHAR(30),
  SSAN        CHAR(11),
  total_sales  MONEY,
  status       VARCHAR(30))

CREATE table customers
(
  customer_number INTEGER,
  company_name     VARCHAR(50),
  salesperson_id   INTEGER)

CREATE table orders
(
  customer_id INTEGER,
  order_number INTEGER,
  order_date   DATETIME,
  order_total  MONEY)
```

可用以下 UPDATE 语句标识定单总数大于\$1,000,000 的单独顾客的销售员：

```
UPDATE employees SET status = 'Key Account Manager'
WHERE employee_id IN
  (SELECT salesperson_id FROM customers
   WHERE customer_number IN
     (SELECT customer_id FROM orders
      GROUP BY customer_id
      HAVING SUM(order_total) > 1000000))
```

注意：并未将 TOTAL_SALES 用作选择标准，因为销售员的 TOTAL_SALES 可大于\$1,000,000，但却没有任何一个顾客的购买总数大于\$1,000,000。例如，如果销售员有 100 个账户，每个都购买了\$20,000 的货物，其 TOTAL_SALES 是\$2,000,000，但没有一位顾客是“关键账户”，因为他们每一个人的购货总数都不超过\$1,000,000。

要理解的重要事情是，在 UPDATE 语句的 WHERE 子句中并不只限于使用目标表。相反，还可使用其列可用作形成选择标准的任何其他表（对该表要有 SELECT 访问权限）。

技巧 57 使用 UPDATE 语句通过视图改变表数据

UPDATE 语句只能有一个目标表，也就是说，UPDATE 语句一次只能改变一个表中的值。由于视图是单一的（虽然是虚拟的）表，因而似乎可使用基于多个底层表的视图使 UPDATE 语句同时修改多个表中的列值。

遗憾的是，DBMS 检查视图以便确保视图的 SELECT 语句只包括单个表。因此，如果试图执行以下语句：

```
UPDATE cust_rep
SET employee_status = 'Terminated', cust_sales_rep = 2
WHERE employee_id = 6
```

其中 CUST_REP 是基于来自多个底层表列的视图，DBMS 将不能执行 UPDATE 语句并用以下错误信息响应该语句：

```
Server: Msg 4405, Level 16, State 2, Line1
View 'CUST_REP' is not updateable because the FROM clause
names multiple tables.
```

在 UPDATE 语句中使用视图代替底层表的两个好处是，可使列名更具描述性并可限制用户可更新的底层表中的列。

例如，假设有一个用以下语句创建的表：

```
CREATE TABLE employees
(emp_id          INTEGER PRIMARY KEY IDENTITY,
fname           VARCHAR(25),
lname           VARCHAR(30),
addr            VARCHAR(30),
SSAN            CHAR(11),
Dept            VARCHAR(20),
Badgno          INTEGER,
Sales           INTEGER,
tot_sales       MONEY,
status          VARCHAR(30),
low_quota       INTEGER,
med_quota       INTEGER,
high_quota      INTEGER,
bonus           INTEGER,
bonus_mult      INTEGER DEFAULT 1)
```

而且只想能够更新在市场部门中的雇员的姓名、地址、社会保险号以及证件号，可使用以下的 CREATE VIEW 语句：

```
CREATE VIEW vw_marketing_sup_emp_update
(employee_number, first_name, last_name, address,
social_security_number, badge_number)
AS SELECT emp_id, fname, lname, addr, ssan, badgno
FROM employees
WHERE dept = 'Marketing'
```

对 VW_MARKETING_SUP_EMP_UPDATE 视图给定 UPDATE 访问权限，用户即可更新（使用 UPDATE 语句）市场部雇员的个人信息以及证件号，而不能改变雇员的部门、状态、计数、容量、限额和奖金信息。

例如，为了指定雇员 123 的证件号（badge_number），市场部的主管可使用以下 UPDATE 语句：

```
UPDATE vw_marketing_sup_emp_update
SET badge_number = 1123
WHERE employee_number = 123
```

当 DBMS 接收到使用视图作为目标表的 UPDATE 语句时，DBMS 就在视图定义中使用 SELECT 语句建立虚拟表。这样一来，在本例中，DBMS 通过从 EMPLOYEES 表中选择市场部的雇员建立临时（虚拟的）表。虚拟（视图）表只有列在视图的 SELECT 语句中的那些列，而且排除了 EMPLOYEE 列的每一行的附加数据。

在建立了视图表（在本例中是 VW_MARKETING_SUP_EMP_UPDATE）之后，DBMS 使

用 UPDATE 语句的 WHERE 子句中的选择标准来决定虚拟 (视图) 表中的哪一行要加以更新 (如果 EMPLOYEES 表中的雇员 123 不在市场部, UPDATE 语句不能执行, 因为雇员 123 并不存在于目标表[VW_MARKETING_SUP_EMP_UPDATE]中)。

最后, DBMS 对底层表中的列进行 UPDATE 语句中的 SET 子句中指定的更新。在本例中, DBMS 将把 EMPLOYEES 表中 EMP_ID 为 123 的行的 BADGNO 列设置为 1123。

读者将在技巧 153–技巧 158 中学习有关创建视图以及 DBMS 如何操作视图的内容。

技巧 58 使用 DELETE 语句从表中删除行

DELETE 语句的句法如下:

```
DELETE from <table name> [WHERE <search condition>]
```

例如, 假设有一个电子邮件数据库, 其中有由下面语句创建的用户名和邮件消息表:

```
CREATE TABLE hotmail_users
(
  user_id VARCHAR(25) PRIMARY KEY,
  name VARCHAR(50),
  address VARCHAR(50),
  phone_number VARCHAR(30),
  password VARCHAR(20)
)

CREATE TABLE hotmail_messages
(
  user_id VARCHAR(25),
  date_time_received DATETIME,
  subject VARCHAR (250),
  sent_by VARCHAR(25),
  date_time_sent DATETIME,
  priority CHAR(1),
  message TEXT
)

CONSTRAINT recipient_account_id
FOREIGN KEY (user_id) REFERENCES hotmail_users)
```

因而, 如果用户 KKI 决定不再继续其电子邮件账户, 就可使用以下 DELETE 语句将其从 HOTMAIL_USERS 表中删除, 而且也从 HOTMAIL_MESSAGES 表中删除其所有的电子邮件消息:

```
DELETE FROM hotmail_messages WHERE user_id = 'KKI'
DELETE hotmail_users WHERE user_id = 'KKI'
```

DELETE 语句中的 WHERE 子句确定 DELETE 语句要从表 (表名紧随关键词 DELETE 之后) 中删除的行或一些行。

由此, 在示例中的第一条 DELETE 语句将从 HOTMAIL_MESSAGES 表中删除多行 (如果 KKI 在文件中有不止一条电子邮件消息的话)。作为对照, 第二条 DELETE 语句将从 HOTMAIL_USERS 表中删除单行, 因为 USER_ID 是惟一的 (在 HOTMAIL_USERS 表的 USER_ID 列上的 PRIMARY KEY 约束指定该表中的所有行在 USER_ID 列必须有惟一的非 NULL 的值)。

注意: 当从多个表中删除行时, 如果表中之中的 PRIMARY KEY 被另一表中的 FOREIGN KEY 所引用时, 执行 DELETE 语句的顺序是重要的。在本例中, HOTMAIL_USERS 表中的 PRIMARY KEY (USER_ID) 被 HOTMAIL_MESSAGES 表的 FOREIGN KEY (RECIPIENT_ACCOUNT_ID) 所引用。因而, 如果试图在删除所有的 HOTMAIL_MESSAGES 表中的 USER_ID 为 KKI 的消息之前删除 HOTMAIL_USERS 表中 USER_ID 为 KKI 的行, 则

例如, 假设公司关闭了 Tulsa 的发货部, 而且想要删除所有的 Tulsa 发货部的工作人员。如果视图名为 VW_TULSA_EMPLOYEES, 且是由以下语句创建的:

```
CREATE VIEW vw_tulsa_employees AS 玩物丧志
SELECT employee_id, first_name, last_name, SSAN, department
FROM employees
WHERE location = 'Tulsa'
```

可使用以下 DELETE 语句:

```
DELETE FROM vw_tulsa_employees WHERE department = 'shipping'
```

从 EMPLOYEES 表中删除那些 LOCATION 列值为 Tulsa 以及 DEPARTMENT 列值为 shipping 的行。

当使用视图作为 DELETE 语句的目标表时, 只能删除那些在视图的 SELECT 子句满足搜索条件的那些行。这样一来, 在本例中, DELETE 语句将删除那些 location 列的值为 Tulsa 的行, 即使位置并未在 DELETE 语句的 WHERE 子句中指定。由此, 工作在其他位置的发货部的雇员仍然留在表内。

简短地说, 只可从目标表中删除存在的行 (也就是说, 如果某行不在表中就不能从表中删除)。因而, 使用视图只能删除那些满足视图的选择标准的行, 因为那些是视图中惟一存在的行。

由于表行是 DELETE 语句可删除的最小单位, 将视图作为目标表的 DELETE 语句将从底层表中删除一整行——即使视图只显示表中的某些列。例如, 假设在本例中的 EMPLOYEES 表是由以下语句所创建的:

```
CREATE TABLE employees
(employee_id INTEGER,
first_name   VARCHAR(25),
last_name    VARCHAR(30),
SSAN         CHAR(11),
location     VARCHAR(20),
department   VARCHAR(20),
total_sales  MONEY)
```

那么以下 DELETE 语句:

```
DELETE FROM vw_tulsa_employees
```

将删除所有那些 LOCATION 列的值为 Tulsa 的行, 即使 VW_TULSA_EMPLOYEES 视图只有 EMPLOYEE_ID、FIRST_NAME、LAST_NAME、SSAN 和 DEPARTMENT 列。

虽然 DELETE 语句删除一整行 (包括未在视图中定义的列), 但可以在 DELETE 语句的 WHERE 子句中只引用底层表中那些为视图定义部分的列。因而, 以下 DELETE 语句:

```
DELETE FROM vw_tulsa_employees WHERE total_sales < 1000
```

将不能执行, 因为 TOTAL_SALES 不是 VW_TULSA_EMPLOYEES 视图 (虚拟表) 中的列。如果试图引用底层表中的未定义为 DELETE 语句目标表 (在本例中是 VW_TULSA_EMPLOYEES 视图) 中的列, DBMS 将用以下错误信息响应:

```
Server: Msg 207, Level 16, State 3, Line 1
Invalid column name 'total sales'
```

第 4 章 处理查询、表达式和总计函数

技巧 61 理解 SELECT 语句的结构

SELECT 语句通常称为查询，因为它告诉 DBMS 回答有关存储在一个或多个数据库表中的数据的问题。例如，如果想要查询（或提问）DBMS 以便得到 Bruce Williams 所下定单的购买日期、货品描述和价值的清单，可执行以下 SELECT 语句：

```
SELECT order_date, description, cost
FROM orders, customers
WHERE ordered_by = customer_id
AND first_name = 'Bruce' AND last_name = 'Williams'
```

SELECT 语句的句法如下：

```
SELECT [ALL | DISTINCT] <select item list>
FROM <table list>
[WHERE <search conditions>]
[GROUP BY <grouping column list>
    [HAVING <having search conditions>]]
ORDER BY <sort specification>]
```

由此，SELECT 语句的各个部分是：

- 关键词 SELECT 后跟想要在 SELECT 语句的结果表中显示的数据项的清单。
- FROM 子句，列出那些列的数据值包括在要显示的数据项清单中或可选的 WHERE 子句搜索标准的一部分的表。
- 可选的 WHERE 子句，其中列出搜索标准，用于选择来自 FROM 子句中列出的表要显示的数据行（如果 SELECT 语句没有 WHERE 子句，DBMS 假设目标表[列在 FROM 子句中的]所有行都满足搜索条件）。
- 可选的 GROUP BY 子句，告诉 DBMS 将根据一个或多个在<grouping column list>中列出的列来组合次级总计查询值（读者将在技巧 200 “使用 GROUP BY 子句根据单一列值组合行”中学习如何使用 GROUP BY 子句）。
- 可选的 HAVING 子句，列出另外的行选择标准，以便根据由 GROUP BY 子句产生的结果（分类汇总）筛选行（读者将在技巧 206 “使用 HAVING 子句筛选包括在组合查询结果表中的行”中学习有关 HAVING 子句的内容）。
- 可选的 ORDER BY 子句，告诉 DBMS 如何将 SELECT 语句结果表中的行分类（如果没有 ORDER BY 子句，DBMS 将以（未分类的）输入表中的顺序来显示数据）。

因而，以下示例的 SELECT 语句：

```
SELECT order_date, description, cost FROM orders, customers
WHERE ordered_by = customer_id
AND first_name = 'Bruce' AND last_name = 'Williams'
```

告诉 DBMS 列出在 ORDER_DATE、DESCRIPTION 和 COST 列中发现的值。

FROM 子句告诉 DBMS, 将在 ORDERS 和 (或) CUSTOMERS 表中查找在 SELECT 子句中列出的列。

同时, WHERE 子句告诉 DBMS 只返回符合以下条件的行(从由 ORDERS 和 CUSTOMERS 通过 CROSS JOIN 运算形成的虚拟表), 即 ORDERED_BY 列(来自 ORDERS 表)包括与 CUSTOMER_ID 列(来自 CUSTOMERS 表)相同值, 而且 FIRST_NAME 列(来自 CUSTOMERS 表)包括值 Bruce 以及 LAST_NAME 列(来自 CUSTOMERS 表)包括值 Williams 的行。

技巧 62 理解处理 SQL 的 SELECT 语句所涉及的步骤

为了让 DBMS 显示表中的值, 只需执行带 FROM 子句的 SELECT 语句, 例如:

```
SELECT * FROM employees
```

显示 EMPLOYEES 表中的所有列和所有行。

在实践中, 几乎所有的 SELECT 语句都包括强制输出数据满足某种标准的 WHERE 子句。另外, 许多 SELECT 语句涉及从多个表选择列数据的问题。

当执行 SELECT 语句时, DBMS 执行以下步骤:

1. 根据 FROM 子句中的一个或多个表创建工作表。如果在 FROM 子句中有两个或多个表, DBMS 将执行 CROSS JOIN 运算来创建<table list>(在 SELECT 语句的 FROM 子句中列出的)中的表的笛卡尔积。例如, 如果 SELECT 语句在其 FROM 子句中列出两个表, DBMS 将创建一个表, 此表由第一个表中的每行与第二个表中的每行连接在一起所组成的。在 DBMS 建立此表之后, CROSS JOIN 运算后的工作表包括所有行的可能组合, 其中行是将来自第一个表的行与来自第二个表中的行结合在一起形成的。

注意:没有一个 DBMS 产品实际上使用 CROSS JOIN 运算构建中间结果表——即使 FROM 子句<table list>中的表只有几行, 生成的表也会太大。例如, 如果有两个 1000 行的表, 则生成的笛卡尔积可能是有 1,000,000 行的表! 要了解的重要事情是, 虽然此表实际上并不存在, 积表却是 DBMS 执行多表查询时的行为的正确的概念性描述。

2. 如果有 WHERE 子句, DBMS 将其搜索条件应用于在步骤 1 中生成的复合(笛卡尔积)表中的每行。DBMS 保留那些搜索条件测试为 TRUE 的行, 而删除那些搜索条件测试为 NULL 或 FALSE 的行。如果 WHERE 子句包括子查询, 则 DBMS 对满足主查询的选择标准的每一个执行子查询。

3. 如果有 GROUP BY 子句, DBMS 将结果表中的行分成多个组, 每个组中的<grouping column list>列具有相同的值。接着, DBMS 将每组减少到单行, 然后将其添加到新的结果表中, 用来代替本步骤开始时的表。

注意:GROUP BY 子句的<grouping column list>中的所有列必须出现在 SELECT 子句中的<select item list>中。

注意:DBMS 将 NULL 看作好像是相等的, 而且将所有的 NULL 值都放入其自己的组中。

4. 如果有 HAVING 子句, DBMS 将其应用于步骤 3 中生成的“组合”表中的每一行。DBMS 保留那些<having search conditions>测试结果为 TRUE 的行, 而删除那些<having search conditions>测试结果为 NULL (未知的)或 FALSE 的行。如果 HAVING 子句有子查询, DBMS 对满足<having search conditions>条件的“组合”表中的每一行执行子查询。

注意:在没有 GROUP BY 子句的 SELECT 语句中不能有 HAVING 子句, 因为 HAVING

子句筛选 GROUP BY 子句的结果。另外，在<having search conditions>中的任何列必须被包括在 GROUP BY 子句的<grouping column list>中。

5. 将 SELECT 子句应用于结果表。如果结果表中的列不在<select item list>中，DBMS 丢弃来自结果表的列。如果 SELECT 子句包括 DISTINCT 选项，DBMS 将从结果表中删除重复的行。

注意：SELECT 子句中的<select item list>可由常数（数值型或字符串）、根据常数或表列的计算结果、列和函数所组成。

6. 如果有 ORDER BY 子句，按<sort specification>所指定的对结果表排序。

7. 对于交互式 SQL SELECT 语句，在屏幕上显示结果表，或者对于编程的 SQL，使用游标将结果表传递到主调（或宿主）程序中。

例如，假设有由以下语句创建的两个表：

```
CREATE TABLE employees
  (emp_id    INTEGER PRIMARY KEY,
   last_name VARCHAR(25),
   trainer   VARCHAR(25),
   sales     INTEGER)
CREATE TABLE sales
  (cust_id   INTEGER PRIMARY KEY,
   sold_by   INTEGER,
   sales_amt MONEY)
```

其中有以下数据：

EMPLOYEES table				SALES table		
id	last_name	trainer	sales	cust_id	sold_by	sales_amt
1	Hardy	Bob	3	1	1	\$6,000
2	Wallace	Greg	3	2	1	\$6,000
3	Green	Bob	2	3	4	\$8,000
4	Marsh	Andy	2	4	2	\$4,000
5	Brown	Greg	0	5	2	\$6,000
				6	3	\$7,000
				7	4	\$4,000
				8	1	\$6,000
				9	2	\$7,000
				10	3	\$9,000

为了得到培训人员在培训销售人员时做得如何的报告，可使用类似如下的查询：

```
SELECT last_name trainer, COUNT(*) AS num_trainees,
       SUM(sales_amt) AS gross_sales, AVG(sales_amt)
FROM employees, sales
WHERE sales > 0
AND emp_id = sold_by
GROUP BY last_name, trainer
HAVING AVG(sales_amt) > 6000
```

在对示例查询执行了 SELECT 语句的步骤 1 之后，DBMS 根据列在 SELECT 语句的 FROM

子句中的两个表（EMPLOYEES 和 ORDERS）中的数据创建下面的 CROSS JOIN 工作表：

CROSS JOIN 工作表

EMPLOYEES			ORDERS			
emp_id	last_name	trainer	sales	cust_id	sold_by	sales_amt

1	Hardy	Bob	3	1	1	\$6,000
1	Hardy	Bob	3	2	1	\$6,000
1	Hardy	Bob	3	3	4	\$8,000
1	Hardy	Bob	3	4	2	\$4,000
1	Hardy	Bob	3	5	2	\$6,000
1	Hardy	Bob	3	6	3	\$7,000
1	Hardy	Bob	3	7	4	\$4,000
1	Hardy	Bob	3	8	1	\$6,000
1	Hardy	Bob	3	9	2	\$7,000
1	Hardy	Bob	3	10	3	\$9,000
2	Wallace	Greg	3	1	1	\$6,000
2	Wallace	Greg	3	2	1	\$6,000
2	Wallace	Greg	3	3	4	\$8,000
2	Wallace	Greg	3	4	2	\$4,000
2	Wallace	Greg	3	5	2	\$6,000
2	Wallace	Greg	3	6	3	\$7,000
2	Wallace	Greg	3	7	4	\$4,000
2	Wallace	Greg	3	8	1	\$6,000
2	Wallace	Greg	3	9	2	\$7,000
2	Wallace	Greg	3	10	3	\$9,000
3	Green	Bob	2	1	1	\$6,000
3	Green	Bob	2	2	1	\$6,000
3	Green	Bob	2	3	4	\$8,000
3	Green	Bob	2	4	2	\$4,000
3	Green	Bob	2	5	2	\$6,000
3	Green	Bob	2	6	3	\$7,000
3	Green	Bob	2	7	4	\$4,000
3	Green	Bob	2	8	1	\$6,000
3	Green	Bob	2	9	2	\$7,000
3	Green	Bob	2	10	3	\$9,000
4	Marsh	Andy	2	1	1	\$6,000
4	Marsh	Andy	2	2	1	\$6,000
4	Marsh	Andy	2	3	4	\$8,000
4	Marsh	Andy	2	4	2	\$4,000
4	Marsh	Andy	2	5	2	\$6,000
4	Marsh	Andy	2	6	3	\$7,000
4	Marsh	Andy	2	7	4	\$4,000
4	Marsh	Andy	2	8	1	\$6,000
4	Marsh	Andy	2	9	2	\$7,000
4	Marsh	Andy	2	10	3	\$9,000
5	Brown	Greg	0	1	1	\$6,000

5	Brown	Greg	0	2	1	\$6,000
5	Brown	Greg	0	3	4	\$8,000
5	Brown	Greg	0	4	2	\$4,000
5	Brown	Greg	0	5	2	\$6,000
5	Brown	Greg	0	6	3	\$7,000
5	Brown	Greg	0	7	4	\$4,000
5	Brown	Greg	0	8	1	\$6,000
5	Brown	Greg	0	9	2	\$7,000
5	Brown	Greg	0	10	3	\$9,000

在步骤 2 中，DBMS 将应用 SELECT 语句的 WHERE 子句中的搜索条件，在本例中其中包括两个判式。第一个判式为 `sales > 0`，将删除那些工作表中的 SALES 列带有 0 值的行；第二个判式 `emp_id = sold_by`，将删除那些 EMP_ID 列的值不等于 SOLD_BY 列的值的行，从而产生以下工作表：

用 WHERE 子句筛选后的 CROSS JOIN 工作表

EMPLOYEES			ORDERS			
emp_id	last_name	trainer	sales	cust_id	sold_by	sales_amt
1	Hardy	Bob	3	1	1	\$6,000
1	Hardy	Bob	3	2	1	\$6,000
1	Hardy	Bob	3	8	1	\$6,000
2	Wallace	Greg	3	4	2	\$4,000
2	Wallace	Greg	3	5	2	\$6,000
2	Wallace	Greg	3	9	2	\$7,000
3	Green	Bob	2	6	3	\$7,000
3	Green	Bob	2	10	3	\$9,000
4	Marsh	Andy	2	3	4	\$8,000
4	Marsh	Andy	2	7	4	\$4,000

注意：如果 WHERE 子句中的任何判式只使用列于 FROM 子句中的一个表，DBMS 的优化器通常在执行 CROSS JOIN 运算以前使用判式从表中删除行。在本例中，优化器已经根据该项考虑删除了雇员 5，因为该雇员 sales 列为 0，因而从工作表中删除了 10 行。

在步骤 3 中，DBMS 将使用 GROUP BY 子句将工作表通过不同的培训人员（trainer 列）分组，然后计算 SELECT 子句中的总计函数。

在用 WHERE 子句筛选和分组之后的 CROSS JOIN 工作表

EMPLOYEES			ORDERS			
emp_id	last_name	trainer	sales	cust_id	sold_by	sales_amt
1	Hardy	Bob	3	1	1	\$6,000
1	Hardy	Bob	3	2	1	\$6,000
1	Hardy	Bob	3	8	1	\$6,000
3	Green	Bob	2	6	3	\$7,000
3	Green	Bob	2	10	3	\$9,000
2	Wallace	Greg	3	4	2	\$4,000
2	Wallace	Greg	3	5	2	\$6,000
2	Wallace	Greg	3	9	2	\$7,000

4	Marsh	Andy	2	3	4	\$8,000
4	Marsh	Andy	2	7	4	\$4,000

总计函数

```
trainer num_trainees gross_sales AVG(sales_amt)
```

```
-----
Bob      5          $34,000          $6,800.0000
Greg     3          $17,000          $5,666.6666
Andy     2          $12,000          $6,000.0000
```

接着, DBMS 将应用 HAVING 子句中的搜索条件来消除平均销售额为 \$6,000.00 或更少的工作表行。

最后, DBMS 将查询结果应用于 SELECT 子句中的 <select item list> 并显示步骤 7 中的查询结果如下:

```
trainer num_trainees gross_sales AVG(sales_amt)
```

```
-----
Bob      5          34000.00          6800.0000
(1 Row(s) affected)
```

当然, 实际的 DBMS 产品在处理查询时并不创建和删除磁盘上的实际的物理表——这对系统资源与处理时间来说可能是非常昂贵的。本技巧中的工作表模仿了 DBMS 执行 SELECT 语句的方式。

技巧 63 使用 SELECT 语句从一个或多个表的行中显示列

最简单的 SQL 查询是显示单表中的列的 SELECT 语句。例如, 如果有一个由以下语句创建的表:

```
CREATE TABLE customer
(
  customer_id INTEGER PRIMARY KEY,
  first_name  VARCHAR(20),
  last_name   VARCHAR(30),
  address     VARCHAR(50),
  phone_number VARCHAR(20)
)
```

可以使用以下 SELECT 语句显示其内容:

```
SELECT * FROM customer
```

查询的 SELECT 子句中的星号 (*) 告诉 DBMS 显示列在 FROM 子句表 (或多个表) 中的所有列。

如果想要显示表中的某些 (而不是所有的) 列, 可在查询的 SELECT 子句的 <select item list> 中只包括那些想要显示的列。例如, 如果想要只显示顾客的 ID、名字和电话号码, 可使用以下 SELECT 语句:

```
SELECT customer_id, first_name, phone_number FROM customer
```

DBMS 将一次一行地处理 FROM 子句中指定的表 (在本例中是 CUSTOMER 表)。在它读取输入表中的每一行时, DBMS 将使用列在 <select item list> 中的列并将其用来创建结果表中的单一行。这样一来, 以下查询:

```
SELECT customer_id, first_name, last_name FROM customers
```

将生成有 4 行的结果表, 其形式如下:

customer_id	first_name	last_name
1	Wally	Cleaver
2	Dolly	Madison
3	Horace	Greely
4	Ben	Stepman

在<select item list>中的所有列名必须是在 SELECT 语句的 FROM 子句列出的表（或多个表）中定义的。例如，如果有两个由以下语句创建的表（CUSTOMERS 和 ORDERS）：

```
CREATE TABLE customers
  (cust_id    INTEGER PRIMARY KEY,
   first_name VARCHAR(20)
   last_name  VARCHAR(30))

CREATE TABLE ORDERS
  (order_number INTEGER PRIMARY KEY,
   order_date   DATETIME,
   cust_id      INTEGER,
   description  VARCHAR(25),
   order_total  MONEY)
```

而且想要查询数据库从而找出所有顾客的姓名、ID 以及日期、描述和总的定单量，可以使用以下 SELECT 语句：

```
SELECT customers.cust_id, first_name, last_name,
       order_date, description, order_total
FROM customers, orders
WHERE customers.cust_id = orders.cust_id
```

其中列值对列在 FROM 子句中的表来说是惟一的，DBMS 将知道要显示的哪个列值来自哪个表——也就是说，FIRST_NAME 和 LAST_NAME 列只出现在 CUSTOMERS 表中。这样一来，当 DBMS 建立结果表中的一行时，知道从 CUSTOMERS 表中获得 FIRST_NAME 和 LAST_NAME 列的值。类似地，ORDER_DATE、DESCRIPTION 和 ORDER_TOTAL 列只出现在 ORDERS 表中，当 DBMS 向结果表中添加行时，知道要从 ORDERS 表中提取这些值。

当想要显示在 FROM 子句中列出的不止一个表中具有相同名称的列时，必须使用合格的列名（<table name>.<column name>），告诉 DBMS 要使用哪个表的数据。在本例中，CUST_ID 出现在 CUSTOMERS 和 ORDERS 两个表中。因此，为了显示来自 CUSTOMERS 表中的 CUST_ID 列的值，在查询的 SELECT 子句中必须将列名键入为 CUSTOMERS.CUST_ID。

类似地，如果 WHERE 子句包括模糊的列名，在本例中也必须在 WHERE 子句中使用合格的列名：

```
WHERE customers.cust_id = orders.cust_id
```

技巧 64 使用 SELECT 语句显示列及计算值

除了使用 SELECT 语句显示列值以外，还可以将其用于显示计算的列和实际值。例如，假设用户管理着一家饭店而且想要获得根据小费的多少获得老主顾对服务的态度的感觉。由于大多数人的小费惯例为，如果服务较好，则小费为服务费的 15%。现在最感兴趣的是那些大

于和小于惯例的小费量。

假如 sales 表是用以下语句创建的：

```
CREATE TABLE sales
  (emp_id    INTEGER,
   meal_total MONEY,
   tip_rec    MONEY)
```

可使用以下 SELECT 语句：

```
SELECT emp_id, meal_total, '* 15% = ',
       meal_total * .15 AS standard_tip, tip_rec,
       tip_rec - (meal_total * .15) AS over_under
FROM sales
ORDER BY over_under
```

生成下面形式的结果表：

emp_id	meal_total		standard_tip	tip_rec	over_under
1	75.3300	* 15% = 11.2995	10.5000		-.799500
2	13.5700	* 15% = 2.0355	1.2500		-.785500
5	89.2500	* 15% = 13.3875	13.5000		.112500
7	110.4800	* 15% = 16.5720	17.2500		.678000
1	125.4400	* 15% = 18.8160	19.7500		.934000
3	271.2200	* 15% = 40.6830	47.5000		6.817000

显示在 EMP_ID、MEAL_TOTAL 和 TIP_REC 列中的数据项是使用 SELECT 语句显示列值的示例（这已在技巧 63 “使用 SELECT 语句从一个或多个表的行中显示列”中学过）。同时 * 15% = 列展示了如何在 SELECT 语句中包括实际的字符串，让其作为列值在 SELECT 语句的结果表中显示出来。

注意：虽然按字面显示字符串可使结果表中的每一行读起来更像一个句子，但 DBMS 将其看作为带有常数值（在本例中为 * 15% =）的新列。当把结果表通过列位置传递回宿主程序时，这种差别是重要的——SELECT 语句中的字面上实际的字符串添加宿主程序“了解”的新列并进行适当地处理。

与实际（常数）字符串列（* 15% =）类似，结果表中的最后两列（STANDARD_TIP 和 OVER_UNDER）并不存在于 SALES 表中。在每一结果表的行中的这些列值是对同一表中的行计算的结果。

本示例展示了列值乘以常数和用列来代替作为数学表达式（如减和乘）中的操作数。简短地说，可以把在自己的 DBMS 实现中可用的任何数学运算或字符串操作函数的结果显示为 SELECT 语句结果表中的列值。

注意：当在数学或字符串操作函数中使用列值时，必须对列的数据类型十分注意。如果试图执行与类型为 CHAR 或 VARCHAR 的列的数学运算（如除法）时，SELECT 语句将不能执行，而且 DBMS 显示一条错误消息。

如果必须混合使用数据类型，如在数值列上追加实际字符串，可使用在自己的 DBMS 中可用的类型转换函数之一来实现。例如 MS-SQL Server，有 STR 函数用于将数字转换为字符串，而 CONVERT 函数将字符串（只包括数字和加号[+]或减号[-]）转换为数字。

技巧 65 使用带 WHERE 子句的 SELECT 语句根据列值选择行

WHERE 子句允许指定想要在 SELECT 语句的结果表中包括输入表中的哪些行。例如，假设想要获得比所有每月销售定额小的所有销售人员的清单。可使用以下 SELECT 语句：

```
SELECT emp_id, first_name, last_name,  
       quota - monthly_sales AS under_by  
FROM employees  
WHERE department = 'SALES' AND monthly_sales < quota
```

来产生以下形式的结果表：

```
emp_id first_name last_name under_by  
-----  
1      Sally      Fields 6  
7      Wally      Wells 9  
9      Bret       Maverick 12
```

WHERE 子句由关键词 WHERE 后跟指定要被提取数据的搜索条件组成。在本例中，WHERE 子句指明，DBMS 要提取那些 DEPARTMENT 列的值为 SALES，而 MONTHLY_SALES 列的值小于 QUOTA 列的值的行。

当处理带 WHERE 子句的 SELECT 语句时，DBMS 对每一行应用搜索条件从而处理输入表。它用表行中的列值代替 WHERE 子句中的列名。在本例中，DBMS 用 DEPARTMENT 列中的值代替 WHERE 子句中的 "department"，而用 MONTHLY_SALES 列中的值代替 "monthly_sales"，用表行的 QUOTA 列的值代替 WHERE 子句中的 "quota"。

在执行了代换之后，DBMS 计算 WHERE 子句。对于 WHERE 子句求值为 TRUE 的那些列的行就包括在结果表中。其列值使 WHERE 子句求值为 FALSE 或 NULL 的行，就排除在结果表外。

因而，可将 WHERE 子句看作筛选器。满足 WHERE 子句中搜索条件的行通过了筛选器。相反，那些不满足搜索条件的行就“粘”在了筛选器上，从而被排除在结果表外。

除了通过将列值与实际的字符串或数字相比较来选择行之外，还可以使用 WHERE 子句通过将列内容与计算值相比较来选择行。

技巧 66 在 WHERE 子句中使用布尔运算符 OR、AND 和 NOT

布尔运算符允许在单一 WHERE 子句中使用多个搜索条件选择行。当想要选择那些满足几个搜索条件之一的行时，可使用 OR 运算符将子句结合成一个复合的搜索条件。作为对照，当想要选择那些满足所有几个搜索条件的行时，可在 WHERE 子句中使用 AND 运算符结合几个搜索条件。最后，如果想要 DBMS 返回那些不满足其标准的行时，可在 WHERE 子句中使用 NOT 运算符引入一个搜索条件。

包括布尔运算符的 SELECT 语句的一般形式如下：

```
SELECT <select item list> FROM <table name>  
WHERE [NOT] <search condition>  
      [<comparison operator> [NOT] < search condition>]...  
      [<last comparison operator> [NOT]  
      <last search condition>]
```


例如,假设有人要买一辆汽车,而且发现了几个可以接受的型号。可以使用以下 SELECT 语句:

```
SELECT year, make, model, cost FROM auto_inventory
WHERE make = 'Jaguar' OR make = 'BMW' OR make = 'Corvette'
```

来返回 AUTO_INVENTORY 表中包括 Jaguars、BMW 和 Corvettes 的清单。OR 运算符组合 WHERE 子句中的搜索条件,告诉 DBMS 显示选择清单中那些搜索条件求值为 TRUE 的行中的列。

如果想要 DBMS 只显示那些所有搜索条件都为 TRUE 的行,可在 WHERE 子句中使用 AND 运算符来组合搜索条件。因而,以下 SELECT 语句:

```
SELECT year, make, model, cost FROM auto_inventory
WHERE make = 'Corvette' AND year > 1990 AND color = 'Red'
```

告诉 DBMS,感兴趣的只是那些 MAKE 列的值为 Corvette, YEAR 列值大于 1990,而且 COLOR 列的值为 Red 的行。如果任何搜索条件之一求值为 FALSE,则 DBMS 从结果表中排除那些行。换一种说法,AND 运算符告诉 DBMS,仅当 WHERE 子句中的所有搜索条件都求值为 TRUE 时,才在结果表中包括该行。

NOT 运算符允许根据在行的列中找不到某些值来选择行。例如,假设你的儿子和女儿刚满 16 岁,可以接受任何一辆汽车,只要投保时的保费每年不超过 \$2,000。为了从 AUTO_INVENTORY 表中得到可接受的车辆的清单,可使用以下 SELECT 语句:

```
SELECT year, make, model, cost FROM auto_inventory
WHERE NOT cost_to_insure > 2000
```

虽然前面的例子都在 WHERE 子句中使用单一类型的布尔运算符,也可在同一 WHERE 子句中组合多个不同的布尔运算符。例如,假设可接受 Jaguar 晚于 1998 年的型号或者任何 Corvette、任何 BMW,只要不是蓝色的。可使用以下 SELECT 语句:

```
SELECT year, color, make, model, cost FROM auto_inventory
WHERE (make = 'Jaguar' AND year > 1998) OR
      (make = 'Corvette') OR
      (make = 'BMW' AND NOT color = 'blue')
```

注意:通常可在搜索条件中使用比较运算符而不使用否定的搜索条件,即由 NOT 运算符引入的搜索条件。例如,如果搜索条件为“不小于”,只要使用大于或等于运算符(>=)即可。类似地,如果搜索条件为“不大于”,可使用小于或等于运算符(<=)代替。使用 NOT 的主要原因是,当直观觉得 NOT 比等价的比较运算符更易于理解时,可使用 NOT。不习惯于数学符号或集合运算的某些人可能对不等于(<>或!=)运算符不熟悉,那么可能会发现以下形式的 WHERE 子句:

```
WHERE NOT <column name> = <value>
```

比

```
WHERE <column name> != <value>
```

或

```
WHERE <column name> <> <value>
```

更易于阅读。

技巧 67 使用 ORDER BY 子句指定由 SELECT 语句返回行的顺序

当 SELECT 语句在回答查询建立其结果表时,并不排列行使其以特定的顺序显示。通常,DBMS

以所选行在输入表中的插入位置的顺序来显示查询结果。为了效率的原因，某些 DBMS 产品在读取输入表中的行时使用一种索引方法。因而，结果表的行可能排列为其出现在索引中的顺序。

如果想要控制行在结果表中出现的顺序，可向 SELECT 语句中添加 ORDER BY 子句。

包括 ORDER BY 子句的 SELECT 语句的一般形式如下：

```
SELECT <select item list> FROM <table name>
```

```
[WHERE <search condition>]
```

```
ORDER BY <sort specification>
```

其中<sort specification>定义如下：

```
<column name | column number> [ASC | DESC]
```

```
[,...<last column name | last column number>
```

```
[ASC | DESC]]
```

可以使用<select item list>中的任何列作为<sort specification>的一部分。

例如，假设想要从最低到最高列出学生成绩。可使用类似于以下的 SELECT 语句：

```
SELECT student_id, last_name, first_name, grade_received
```

```
FROM students
```

```
ORDER BY grade_received
```

这将生成以下形式的结果表：

```
student_id last_name first_name grade_received
```

```
-----
```

```
1      Smith      Sally      65
```

```
8      Wells      Wally      70
```

```
9      Luema      Albert     75
```

```
12     Luema      Abner      75
```

```
90     Davis      Scott      96
```

如果在排序规格中忽略了关键词 ASC（升序）和 DESC（降序），则 DBMS 将以升序对结果表中的行排序。因而，在本例中，DBMS 根据保存在 GRADE_RECEIVED 列中的值对结果表以升序排序。

为了对不止一行的结果表排序，只要在<sort specification>中包括所有想用于排序的列（请记住，对列在<sort specification>中的列的惟一限制是，它们必须作为列名都出现在 SELECT 语句的<select item list>中）。

因而，为了对成绩单以 GRADE_RECEIVED 列值的降序排序，而对姓名以升序排序，最后对 STUDENT_ID 以升序排序，可使用以下 SELECT 语句：

```
SELECT student_id, last_name, first_name, grade_received
```

```
FROM students
```

```
ORDER BY grade_received DESC, last_name ASC, first_name,
```

```
student_id
```

这将产生以下形式的结果表：

```
student_id last_name first_name grade_received
```

```
-----
```

```
90     Davis      Scott      96
```

```
12     Luema      Abner      75
```

```
9      Luema      Albert     75
```

8	Wells	Wally	70
1	Smith	Sally	65

列在 ORDER BY 子句中的第二和其后的列的作用为附属判据。如果（正如本例中的情况一样）两行在列在 ORDER BY 子句的第一列上有相同的值（在本例中有两行的 GRADE_RECEIVED 列都为 75），DBMS 首先比较列在 ORDER BY 子句的第二列上的值来决定哪一行排在先。如果第二列也是一样的（在本例中的两个 LAST_NAME 列都有值 Luema），DBMS 将比较列在 ORDER BY 子句的第 3 列中的值等。

请注意，在同一 ORDER BY 子句中，可以混合使用升序和降序的排序顺序。很明显，单列既可以升序也可以降序排列，但不能同时以两种顺序排列。但是，正如本例中的情况，如果在 ORDER BY 子句中列出多列，则其中每一个都既可以升序（ASC）也可以降序排列，而不用管另一列如何。

除了在 ORDER BY 子句中使用列名之外，还可通过数字引用 <select item list> 中的项目。例如，为了对前例结果表中的结果以 GRADE_RECEIVED、LAST_NAME、FIRST_NAME 和 STUDENT_ID 列排序，可在 SELECT 语句中按以下方式使用 ORDER BY 子句：

```
SELECT student_id, last_name, first_name, grade_received
FROM students
ORDER BY 4 DESC, 2, 3, 1
```

而代替下面的形式：

```
SELECT student_id, last_name, first_name, grade_received
FROM students
ORDER BY grade_received DESC, last_name ASC, first_name,
        student_id
```

ORDER BY 子句中的列号是由其在 <select item list> 中的位置，而不是在输入表中的位置决定的。因而，在本例中，可将 <select item list> 中的第四个项目 GRADE_RECEIVED 列用 4 来引用，而不管 GRADE_RECEIVED 列在 STUDENTS 表中是定义为第 1 列、第 10 列或是第 15 列。

当想要用在输入表中并不存在或是没有列名的计算的列来排序时，就可在 ORDER BY 子句中使用列号来代替列名。例如，为了以超过（或未达到）定额的销售数字来显示 EMPLOYEES 表中的销售人员时，可使用以下 SELECT 语句：

```
SELECT employee_id, first_name, last_name, quota,
        sales, sales - quota
FROM employees
ORDER BY 6 DESC, last_name, first_name, employee_id
```

DBMS 将返回以下形式的结果表：

emp_id	first_name	last_name	quota	sales	
1	Sally	Fields	3	7	4
7	Wally	Wells	8	9	1
9	Bret	Maverick	7	5	-2

注意：如果在选择项目清单中的销售定额项上添加 AS 子句，就可在 ORDERED BY 子句中使用自己给计算的列起的名字。例如，如果执行以下 SELECT 语句（其中将 SALES-QUOTA 这个计算值命名为 over_under）：

```
SELECT employee_id, first_name, last_name, quota,
```

```

        sales, sales - quota AS over_under
FROM employees
ORDER BY over_under DESC, last_name, first_name,
employee_id

```

DBMS 将显示以下结果表：

```

emp_id first_name last_name quota sales over_under
-----
1      Sally      Fields      3      7      4
7      Wally      Wells      8      9      1
9      Bret      Maverick    7      5     -2

```

请注意，结果表仍然以与以前同样的顺序显示，但在每一行末尾的计算的列现在有了标号 over_under。

技巧 68 在 WHERE 子句中使用复合条件 (AND、OR 和 NOT) 根据多个列值（或计算值）选择行

布尔运算符 AND 和 OR 允许将多个单独的搜索条件结合在一个 WHERE 子句中形成一个复合的搜索条件。同时 NOT 运算符允许对搜索条件的求值结果求反，从而告诉 DBMS 选择搜索条件为 FALSE（非 TRUE）的行。

当可能有多个条件为 TRUE，但只有一个必须为 TRUE，才能选择那些要包括在结果表中的行时，可使用 OR 运算符来组合搜索条件。例如，为了生成有资格参加美国高尔夫球公开锦标赛的高尔夫球员的表，可使用以下 SELECT 语句：

```

SELECT first_name, last_name FROM golfers
WHERE previous_us_open_winner = 'Y' OR
      PGA_tournaments_won > 1 OR
      qual_school_ranking <= 10

```

当球员曾是上届美国高尔夫球公开锦标赛的获胜者或者当球员是其他 PGA 锦标赛的获胜者，或者当球员以前 10 名成绩完成了其在合格学校的学业时，就允许该球员参赛。

当对复合搜索条件求值时，DBMS 对每个单独的搜索条件求值，然后执行布尔数学运算来决定整个 WHERE 子句是求值为 TRUE 还是 FALSE。这样一来，在本例中，DBMS 将对 3 个搜索条件（PREVIOUS_US_OPEN_WINNER='Y'、PGA_TOURNAMENTS_WON > 1 和 QUAL_SCHOOL_RANKING <= 10）中的每一个求值，然后使用表 4.1 中的 OR 真值表来决定 SELECT 语句中的 WHERE 子句是 TRUE 还是 FALSE。

表 4.1 OR 真值表

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

如果 WHERE 子句求值为 TRUE，DBMS 就在结果表的当前行中包括 <select item list> 值，如果 WHERE 子句求值为 FALSE 或 NULL，DBMS 就排除该值。

AND 运算符允许形成复合搜索条件，通过要求列在 WHERE 子句中的所有搜索条件都为

TRUE, 从而使整个 WHERE 子句求值为 TRUE。例如, 如果要求销售业绩大于\$75,000, 且其定单取消率小于 10%, 而且销售次数大于 30 的销售人员才有资格获得奖金, 可使用以下 SELECT 语句生成获奖资格人员表:

```
SELECT employee_id, first_name, last_name, '$200' AS bonus
FROM employees
WHERE department = 'Sales' AND
      gross_sales > 75000 AND
      (cancellations / sales) < .1 AND
      sales > 30
```

正如 OR 运算符的情况一样, DBMS 将对单独的搜索条件求值(在本例中有 4 个搜索条件), 然后使用表 4.2 中的 AND 真值表来决定 WHERE 子句是 TRUE 还是 FALSE。

表 4.2 AND 真值表

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

如果 WHERE 子句求值为 TRUE, 则 DBMS 将来自当前行的<select item list>值包括在结果表中, 如果 WHERE 子句求值为 FALSE 或 NULL, 则 DBMS 排除该值。

最后, 可以使用 NOT 运算符对搜索条件的布尔值(TRUE 或 FALSE)求反。例如, 如果想要获得那些销售毛额不大于\$75,000 或是未达到 30 次销售次数的销售人员清单, 可使用以下 SELECT 语句:

```
SELECT employee_id, first_name, last_name, gross_sales,
       sales
FROM employees
WHERE department = 'Sales' AND
      ((NOT gross_sales > 75000) OR
      (NOT sales > 30))
```

当决定搜索条件的求反值时, DBMS 使用表 4.3 中 NOT 真值表中给出的逻辑。

表 4.3 NOT 真值表

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

技巧 69 理解使用比较判式选择行时的 NULL 值

当编写搜索条件时, 一定要考虑处理 NULL 值。因为结果表将只包括那些搜索条件为 TRUE 的行, 因而, 即使显示了满足搜索条件的行, 然后又显示了满足搜索条件求反的行, 某些表行将仍然是“隐藏的”。

例如, 假设有一个有以下值的雇员表:

```
first_name last_name sales quota
-----
```

Sally	Fields	8	5
Wally	Wells	4	10
Sue	Smith	10	NULL
Kelly	Sutherland	7	7

以下 SELECT 语句将产生只有一个人名 Wally 的结果表：

```
SELECT first_name FROM employees WHERE sales < quota
```

作为对照，以下 SELECT 语句将产生有两个人名 Sally 和 Kelly 的结果表：

```
SELECT first_name FROM employees WHERE sales >= quota
```

向两个结果表中添加行（销售小于定额和等于或大于定额的雇员），人们可能会作出错误的结论，EMPLOYEES 表中有 3 个销售人员，但事实上有 4 个。

一个销售人员从视线中消失的原因是，无论使用哪个比较运算符（=、>、<、<>、>=、<=），如果被比较的列是 NULL，则整个表达式将成为 NULL。因而，该行将排除在结果表外，因为结果表包括那些搜索条件为 TRUE 的行，那些求值为 FALSE 和 NULL（未知的）的行都已被排除。所以，在本例中，雇员 Sue 绝不会出现在根据对定额列（对 Sue 来说其值为 NULL）作为操作数使用搜索条件而生成的结果表中。

甚至以下 SELECT 语句：

```
SELECT first_name FROM employees WHERE quota = NULL
```

将生成带有零行的结果表！（如果使用比较运算符的表达式中的任何操作数为 NULL，则 DBMS 将整个表达式求值为 NULL，并将其排除在结果表之外。）

注意：为了获得搜索条件求值为 NULL 的行的清单，可使用 NULL 值测试 IS NULL。因此，为了获得在 QUOTA 列有 NULL 值的雇员清单，可使用以下 SELECT 语句：

```
SELECT first_name FROM employees WHERE quota IS NULL
```

技巧 70 使用行值表达式根据多个列值选择表中的行

顾名思义，行值构建器是实际值和表达式的列表，两者一起给出表中单一行中的列值。在技巧 47“使用 INSERT 语句向表中添加行”中，读者在向表中添加行时曾经学习了有关行值构建器方面的内容。

类似于在 INSERT 语句中使用的行值构建器，SELECT 语句中的行值构建器指定行中列的值。但是，与 INSERT 语句中的行值构建器不同，SELECT 语句的 WHERE 子句中的行值构建器并不用于向表中添加行。而是把 SELECT 语句的 WHERE 子句中的行值构建器用于指定行中的列必须有的值，以便该行被包括在 SELECT 语句的结果表中。

例如，为了显示销售部门中活跃的雇员，可使用以下 SELECT 语句：

```
SELECT employee_id, first_name, last_name, gross_sales
FROM employees
WHERE (status, department) = ('Active', 'Sales')
```

为了处理带有行值构建器的 SELECT 语句，DBMS 一次一行地处理输入表，使用在行值构建器中指定的每列的值来代替等号（=）左边的列名。DBMS 根据等号左边代替的列值并将其与行值构建器等号右边的值清单中给定的实际值（或表达式）构建的行比较来构建一行。

通过比较两行中的每一对列来完成比较，也就是说，等号左边的清单中的第一个值与等号右边的清单中的值成为一对并比较，等号左边的第二个值与等号右边的第二个值成对并比

较，如此进行下去。

仅当每一对列具有相同值时，行值比较是否相等(如在本例中的情况那样)的结果就是 TRUE。

注意：并非所有的 DBMS 产品都在 SELECT 语句的 WHERE 子句中支持行值表达式。如果 DBMS 不支持，可用带复合搜索条件的 WHERE 子句改写行值表达式。例如，在本例中的 SELECT 语句可改写为：

```
SELECT employee_id, first_name, last_name, gross_sales
FROM employees
WHERE status = 'Active' AND department = 'Sales'
```

正如读者在本技巧中所学习的，在行值构建器的等号左边的值清单中的每个列值与等号右边的值清单中的每个实际值成对并比较。因而，可将行值等式重新构建为用 AND 运算符组合在一起的一系列“成对”的列对值的等式搜索条件，因为所有（而不是某些）的搜索条件必须求值为 TRUE，才能使 WHERE 子句求值为 TRUE。

技巧 71 理解子查询

子查询是 SELECT 语句内的一条 SELECT 语句，而且常常被称为内查询或是内 SELECT 语句。作为对照，包括子查询的 SELECT 语句被称为外查询、外 SELECT 语句或主 SELECT 语句。子查询常常用在 SELECT 语句的 WHERE 子句中用于生成一个单列的虚拟表，主 SELECT 语句用此值来确定哪些输入表列应包括在结果表中。例如，假设想要所有 PRODUCTS 表中那些在 ORDERS 表中未发货定单上的数量大于当前存货量的产品的清单，可使用以下形式的 SELECT 语句来生成后备定单货品的清单，只要知道总的要发货的值即可：

```
SELECT product_id, description, qty_in_stock FROM products
WHERE qty_in_stock < "total to ship"
```

以下形式的查询：

```
SELECT SUM(quantity) FROM orders
WHERE orders.item_number = "item to total" AND
      date_shipped IS NULL)
```

将返回其货品号用来代替货品总数的每种货品到期要发货的数量（假定未发货定单在 DATE_SHIPPED 列的值为 NULL）。

作为子查询将第二个 SELECT 语句添加到第一个上，即可得到：

```
SELECT product_id, description, qty_in_stock FROM products
WHERE qty_in_stock < (SELECT SUM(quantity)
                      FROM orders
                      WHERE orders.item_number =
                          products.product_id AND
                          date_shipped IS NULL)
```

当执行这个新的带有子查询的 SELECT 语句时，DBMS 处理 PRODUCTS 表的每一行并执行子查询，以便确定到期要发货的数量。每个子查询的执行产生一个包括单一值的虚拟表，DBMS 将这一值与 QTY_IN_STOCK 进行比较。只要存货量（QTY_IN_STOCK）小于到期要发货的数量（由子查询所返回的），DBMS 将把来自 PRODUCTS 表当前行的 PRODUCT_ID、DESCRIPTION 和 QTY_IN_STOCK 列的值包括在结果表中。

子查询的句法如下：

```
(SELECT [ALL | DISTINCT] <select item list>
FROM <table list>
[WHERE <search condition>]
[GROUP BY <group item list>
[HAVING <group by search condition>]])
```

这样一来，除了被括在括号()中之外，子查询看起来（其功能也是）完全与其他 SELECT 语句一样。但是，有几个 SELECT 语句作为子查询必须遵守的规则：

- 如果子查询是为了向 WHERE 子句的比较运算符（=、<>、>、>=、<、<=）提供值，子查询大多数情况下返回单值，也就是说，子查询的结果表必须由带有至少一行的单列组成（一行也没有的子查询求值为 0，而不是错误条件）。
- 如果子查询是由关键词 IN 或 NOT IN 引入的，则子查询必须返回一列的结果表，但是，结果表可以有不止一行。
- 不能有 ORDER BY 子句（因为子查询的结果表并不向用户显示而且用作其外 SELECT 语句的输入表，对子查询的结果表排序没有实际意义）。
- 出现在子查询中的列名既可在列在子查询的 FROM 子句中的表中定义，也可在列在外 SELECT 语句的 FROM 子句中的表中定义。
- 在大多数 DBMS 执行中，子查询只可由一条 SELECT 语句组成，而且不能由几条 SELECT 语句经 UNION 联合组成。

在本技巧中，读者学习了如何使用子查询来生成一个 DBMS 用作主 SELECT 语句的 WHERE 子句中的搜索标准的（虚拟的）值表。技巧 250“理解 HAVING 子句中子查询的作用”将向读者展示如何使用子查询在 SELECT 语句的 HAVING 子句中生成筛选器值。

技巧 72 使用行值子查询根据多个列值选择表中的行

行值子查询是内 SELECT 语句，其结果表（与读者在技巧 70“使用行值表达式根据多个列值选择表中的行”中学过的行值表达式类似）由不止一列组成。在技巧 71“理解子查询”中，读者学习了对于用于向比较运算符提供值的子查询的限制之一是，它必须返回单列。但是，这只是因为大多数 DBMS 产品不支持 SQL-92 标准中的行值表达式时才是正确的。

包括在 SQL-92 标准中的行值运算之一是行值比较。如果 DBMS 支持行值比较，子查询可返回多列数据，DBMS 将其与一个行值构建器中的值加以比较。

例如，假设想要获得那些购买了汽车存货目录中最高价格汽车的顾客的 ID 号。首先需要提交一个如下的子查询，以便确定存货目录中最高价格的汽车的制造商、样式和型号：

```
SELECT manufacturer, make, model FROM auto_inventory
WHERE sticker_price = (SELECT MAX (sticker_price)
                       FROM auto_inventory)
```

然后让 DBMS 将所购买的每辆汽车的 MANUFACTURER、MAKE 和 MODEL 与最高价格的汽车的 MANUFACTURER、MAKE 和 MODEL 进行比较，将示例的 SELECT 语句作为子查询包括在以下 SELECT 语句中：

```
SELECT customer_id, order_date, price_paid
FROM auto_purchases
WHERE (manufacturer, make, model) =
      (SELECT manufacturer, make, model FROM auto_inventory
```



```
WHERE sticker_price = (SELECT MAX (sticker_price)
                        FROM auto_inventory))
```

在本例中，WHERE 子句的等号右边包括一个子查询，当执行时将返回 3 列：汽车存货目录中最高价格的汽车的 MANUFACTURER、MAKE 和 MODEL 列。通过将子查询的结果与等号左边的行值构建器中的值加以比较，DBMS 即可决定是否要包括来自外查询结果表中的 AUTO_PURCHASES（输入）表中的 CUSTOMER_ID、ORDER_DATE 和 PRICE_PAID 列值。

如果由等号右边的子查询所返回的列值等于等号左边的行值构建器中的列值，DBMS 将包括 SELECT 语句结果表中的 AUTO_PURCHASES 表的当前行中的 CUSTOMER_ID、ORDER_DATE 和 PRICE_PAID 值。

如果 DBMS 不支持行值构建器作为比较中的操作数，可将多列比较分成由 AND 运算符连接起来的单列搜索条件。在本例中，可把带行值子查询的 SELECT 语句改写为以下的 SELECT 语句：

```
SELECT customer_id, order_date, price_paid
FROM auto_purchases
WHERE manufacturer = (SELECT DISTINCT manufacturer
                      FROM auto_inventory
                      WHERE sticker_price =
                        (SELECT MAX (sticker_price)
                         FROM auto_inventory))
AND make = (SELECT DISTINCT make FROM auto_inventory
            WHERE sticker_price =
              (SELECT MAX (sticker_price)
               FROM auto_inventory))
AND model = (SELECT DISTINCT model FROM auto_inventory
             WHERE sticker_price =
               (SELECT MAX (sticker_price)
                FROM auto_inventory))
```

在本示例中的查询使用了 3 个标量子查询来产生与由前面的 SELECT 语句相同的结果表，后者使用了单行值构建器来做出同样的查询。

注意：“标量”子查询是那种返回单值的查询（也就是说，子查询的结果表是由单列和至少单行组成）。如果标量查询的结果表没有行，那么其值为 0。在本例中假设每辆汽车的最大滞销货价格对制造商、样式和型号来说都是惟一的。例如，如果不止一种型号其滞销货价格等于最大滞销货价格，SELECT DISTINCT MODEL 子查询将不再返回标量，而且该 SELECT 语句将不能执行。

比较本例中的最后两个 SELECT 语句的长度和结构，可以看出，带行值子查询的 SELECT 语句更为简洁而且易于理解，因为它更紧密地遵循查询的英语描述。

技巧 73 理解表达式

SQL-92 标准定义了 4 种可用于表达式中的算术运算符：加（+）、减（-）、乘（*）和除（/）。另外，SQL-92 标准允许使用括号（()）将运算符和操作数组合成复杂的表达式。

数学运算符的顺序或优先级大家是熟悉的：先乘除后加减。因而，为了对表达式 $A + B / C - D * E$ 求值，DBMS 将：

1. B 除以 C。
2. D 乘以 E。
3. 将步骤 1 的结果加到 A 上。
4. 从步骤 3 的结果减去步骤 2 的结果。

在本例中从左向右读取方程，人们或许认为 DBMS 会将 A 与 B 首先相加，然后将和除以 C。但是，优先级的数学顺序表明，除和乘要先执行，然后才是加和减运算。

如果想要改变运算的顺序或是使表达式易于阅读，可使用括号将运算组合在一起。例如，如果将表达式 $A + B / C - D * E$ 改写为 $A + (B / C) - (D * E)$ ，就更容易看出哪个操作数和结果将会受到数学运算的影响。另外，如果想要改变运算的顺序，以便使其从左向右（正如阅读的顺序）进行，可将该表达式改写为 $((A + B) / C - D) * E$ 。

在 SQL 中，表达式有 3 种功能：

- 为了计算要在 SELECT 语句的结果表中返回的值。例如，为了列出雇员及其工作一小时所生成的收入和每付出一美元工资的收入，可使用以下 SELECT 语句：

```
SELECT employee_id, first_name, last_name, gross_sales,  
       gross_sales / hours AS revenues_per_hour,  
       gross_sales / amount_paid AS  
       revenues_per_dollar_paid  
FROM employees
```

- 为了计算用于搜索标准中的值。例如，为了显示那些销售额比定额的 50% 还低的雇员，可使用以下 SELECT 语句：

```
SELECT employee_id, first_name, last_name, quota, sales  
FROM employees WHERE sales < .50 * quota
```

- 为了计算用于更新表中数据的值。例如，为了将所有高级代表的销售定额增加 25%，可使用以下 SELECT 语句：

```
UPDATE employees SET quota = quota * .25  
WHERE department = 'Marketing' and status = 'Senior'
```

注意：虽然 SQL-92 只指定 4 种数学运算符，但 DBMS 很可能包括三角函数（SIN、COS、TAN 等）和其他运算符，如指数、舍入、平方根等。请检查一下系统文档，从而得到自己的 DBMS 所支持的数学运算的全部清单。

技巧 74 理解 SQL 的判式

ANSI/ISO 标准将搜索条件称为判式。由此，读者已经知道什么是判式以及如何使用判式，因为已经在具有 WHERE 子句的 SQL 语句的示例技巧中看到了判式。

例如，以下 SELECT 语句：

```
SELECT student_id, first_name, last_name FROM students  
WHERE gpa = 4.0
```

使用了比较判式 $gpa = 4.0$ 告诉 DBMS 在 STUDENTS 表中筛选或是选择那些 GPA 列的值为 4.0 的行。

判式是 WHERE 的一部分（或搜索条件），描述了 DBMS 感兴趣的数据值。由于 SQL 是数据子语言，是为描述所要的数据而不描述如何提取和存储数据而设计的。读者将在几乎所有的 SELECT、UPDATE 和 DELETE 语句中发现 SQL 判式（向 DBMS 描述数据）。

有 7 种基本的判式种类：

- 比较判式（=、<>、>、>=、<、<=），用于比较两个表达式或表列的值。
- 范围测试判式（BETWEEN），用于检查表达式的值或表列的值是否落在两个值（范围的上限和下限）之间。范围包括端点。
- 集合成员判式（IN），测试表达式或表列的值是否可在一个值的列表（或集合）中找到。
- 模式匹配判式（LIKE），用于检查 CHARACTER 类型的列值是否有与指定字符模式相匹配的字符串。
- NULL 值判式（IS NULL），用于检查一列是否包括 NULL 或未知的值（某些 DBMS 产品使用关键词 ISNULL 代替 IS NULL）。
- 量词判式（ALL、SOME、ANY），允许使用比较判式将一列的值或表达式的值与在集合或值的列表中的值比较。
- 存在性判式（EXISTS），与子查询结合使用，用以确定一个表内是否包括任何满足子查询搜索条件的行。

当执行带有判式的 SQL 语句时，DBMS 执行对判式求值为 TRUE 的表行进行操作，而对判式求值为 FALSE 或未知的表行不采取任何行动。

技巧 75 理解集合（或列）函数

除了从表的单独的行中提取数据之外，有时需要基于看作集合的一列中的所有值的信息。SQL-92 具有 6 个集合或列函数，可完成以上任务（集合[或列]函数有时又称为总计函数，因为它们对存储在多行中的一列中的值总计或累加成单一值）。

SQL 的集合（总计）函数有：

- COUNT(*), 返回满足 WHERE 子句中的搜索条件的表行的数目
- COUNT(<column name>), 返回列中非 NULL 值的数目
- MAX, 返回表列中的最大值
- MIN, 返回表列中的最小值
- SUM, 返回列中值的总和
- AVG, 返回一列中值的平均值

例如，为了对 EMPLOYEES 表中的雇员计数，可使用以下 SELECT 语句：

```
SELECT COUNT(*) FROM employees
```

或者，为了显示居住在 Nevada（缩写为 NV）的雇员数，可使用以下 SELECT 语句：

```
SELECT COUNT(*) FROM employees WHERE state = 'NV'
```

当需要在特定的列中的非 NULL 值的表行计数时，可用 SELECT (<column name>) 总计函数代替 SELECT (*) 函数。例如，以下 SELECT 语句将显示居住在 Nevada 且在 QUOTA 有非 NULL 值的雇员数：

```
SELECT COUNT(quota) FROM employees WHERE state = 'NV'
```

以下两语句：

```
SELECT COUNT(*) FROM employees WHERE state = 'NV'
```

和

```
SELECT COUNT(quota) FROM employees WHERE state = 'NV'
```

之间的差别是，第二条语句并不计数任何在<QUOTA>字段有 NULL 值的行。因而，当需要对满足搜索标准的行计数时，可使用 SELECT COUNT(*)。当想要只对满足搜索条件且在特定的列上有非 NULL 值的行计数时，可使用 SELECT COUNT (<column name>)语句。

为了确定一列中的最大值和最小值，可使用 MIN 或 MAX 函数。例如，为了显示最老的雇员的 ID 和姓名，可使用以下 SELECT 语句：

```
SELECT employee_id, first_name, last_name, age
FROM employees
WHERE age = (SELECT MAX(age) FROM employees)
```

作为对照，如果想要得到最年青的雇员的清单，可在 SELECT 语句中使用 MIN 函数：

```
SELECT employee_id, first_name, last_name, age
FROM employees
WHERE age = (SELECT MIN(age) FROM employees)
```

当需要把一列中的值都加起来时，可使用 SUM 函数，例如，为了显示在 2000 年 5 月的销售额，可使用以下 SELECT 语句：

```
SELECT SUM (order_total) FROM sales
WHERE date_sold >= '05/01/2000' AND
      date_sold <= '05/31/2000'
```

为了对一列中的值求和，该列当然必须定义为数值数据类型之一。另外，列中值的总和结果必须落在该数据类型的范围之内。因而，如果试图确定类型为 SMALLINT 的列的总和，生成的总数不能大于 SMALLINT 数据类型的上限。

AVG 函数返回在数值数据类型列中值的平均。例如，除了显示 2000 年 5 月的销售总量之外，还要显示定单价值的平均数，可向前面的 SELECT 语句中添加 AVG 函数：

```
SELECT SUM (order_total), AVG (order_total) FROM sales
WHERE date_sold >= '05/01/2000' AND
      date_sold <= '05/31/2000'
```

注意：根据定义，NULL 具有不可确定的值，NULL 的值是“未知的”。因而，带有 NULL 的任何行既被 SUM 函数忽略也被 AVG 函数忽略，对一列中计算的总和或平均没有作用。

技巧 76 理解 CASE 表达式

SQL 是一种数据子语言，而不是全功能的编程语言。由于 SQL 是为允许指定想要的数据库而设计的，而不是为如何获得数据库而设计的，原始的 SQL 规范并不包括块 (BEGIN、END) 语句、条件语句、分支 (GOTO) 语句或循环 (DO、WHILE、FOR) 语句。为了减少对外部源程序操作中间查询结果的依赖，许多 DBMS 开发商已经向 SQL 中添加了块语句和流程控制语句 (MS-SQL Server 扩展称为 Transact-SQL，而 Oracle 扩展可在 SQL*PLUS 和 PL/SQL 中找到)。一种通用的“编程语言”结构——CASE 表达式——甚至已经出现在 SQL-92 规范中。

CASE 表达式与几乎所有编程语言中都有的 IF-THEN-ELSE 语句在功能上类似的。CASE 表达式减少了 SQL 在处理结果表数据时对外部程序的依赖性，给予数据子语言以有限的作出

决定的能力。因此,如果需要,不必在提取数据之后再运行独立的外部程序来修改输出数据(结果表),修改的结果可由当前行或总计数据值来决定,而且只要那些变化由当前表行中的列值所触发。

例如,如果有由以下语句定义的雇员表:

```
CREATE TABLE employees
  (id          CHAR(3),
   name        VARCHAR(35),
   address     VARCHAR(45),
   phone_number CHAR(11),
   department  SMALLINT,
   commission  MONEY,
   bonus_level VARCHAR(35),
   total_sales MONEY,
   hourly_rate MONEY,
   sales_calls SMALLINT,
   sales_count SMALLINT)
```

可使用以下 SELECT 语句显示表的内容:

```
SELECT id, name, department FROM employees
ORDER BY department
```

表的内容如下:

id	name	department
3	William Silverman	1
4	Walt Welinski	1
1	Carry Grant	2
2	Michael Lancer	2
5	Sally Fields	3
6	Walt Frazier	3
7	Melissa Gomez	4

如果想要将部门号转化为字符(串)描述,在 SQL-92 之前,可能必须使用外部程序来转化数据。SQL-92 的 CASE 表达式允许根据表行中的列值即时修改输出。

SQL-92 的 CASE 表达式有两种形式,这依赖于 CASE 表达式是“简单的”还是“搜索的”而定。“简单的”CASE 表达式基于关键词 CASE 后面的值和 CASE 表达式中的关键词 THEN 之后的值之间的直观相等。同时“搜索的”CASE 表达式关键词 CASE 之后的表达式中有比较运算符。

“简单的”CASE 表达式句法基于<value to test>(跟在关键词 CASE 之后)等于每个关键词 WHEN 之后的<value>:

```
CASE <value to test>
  WHEN <value> THEN <expression> | NULL
  [WHEN <value> THEN <expression> | NULL]...
  [WHEN <last value> THEN <last expression> | NULL]
  [ELSE <expression> | NULL]
END
```

这样一来，为了将本例中的结果表中的部门号转化为实际的字符串，可使用以下 SELECT 语句：

```
SELECT id, name, CASE department
                WHEN 1 THEN 'Marketing'
                WHEN 2 THEN 'Customer Service'
                WHEN 3 THEN 'Collections'
                WHEN 4 THEN 'Customer Relations'
                END AS dept_name
FROM employees
ORDER BY dept_name
```

产生的结果表如下：

id	name	dept_name
5	Sally Fields	Collections
6	Walt Frazier	Collections
7	Melissa Gomez	Customer Relations
1	Carry Grant	Customer Service
2	Michael Lancer	Customer Service
3	William Silverman	Marketing
4	Walt Welinski	Marketing

“搜索的” CASE 表达式的句法如下：

```
CASE WHEN <search condition> THEN <expression> | NULL
      [WHEN <search condition> THEN <expression> | NULL]...
      [WHEN <last search condition> THEN
        <last expression> | NULL]
      [ELSE <expression> | NULL]
END
```

而且可将本例中的简单 CASE 表达式改写为搜索 CASE 表达式：

```
SELECT id, name,
       CASE WHEN department = 1 THEN 'Marketing'
            WHEN department = 2 THEN 'Customer Service'
            WHEN department = 3 THEN 'Collections'
            WHEN department = 4 THEN 'Customer Relations'
       END AS dept_name
FROM employees
ORDER BY dept_name
```

虽然可使用搜索的 CASE 表达式来检查列的内容是否等于特定的值，但搜索的 CASE 表达式的实际能力是，允许在测试某事而不是相等的搜索条件中使用子查询和比较运算符。例如，假设为销售人员分配奖金池，根据总销售量增加佣金比例。可使用以下带搜索的 CASE 表达式的 SELECT 语句来根据销售人员的 TOTAL_SALES 落入的奖金范围来显示其 ID、姓名、和奖金池：

```
SELECT id, name, total sales,
       CASE WHEN total_sales < 10000 THEN 'Rookie'
            WHEN (total_sales >= 10000) AND
```

```
        (total_sales < 100000) THEN 'Associate'
    WHEN (total_sales >= 100000) AND
        (total_sales < 1000000) THEN 'Manager'
    WHEN (total_sales >= 1000000) THEN
        'Vice President'

    END
FROM employees
ORDER BY name, id
```

技巧 77 使用 CASE 表达式更新列值

SQL 的 CASE 表达式与在大多数编程语言中的 CASE 语句不同，因为它只能用作 SQL 语句的一部分，其自己不能成为一条语句。换句话说，SQL 的 CASE 表达式可出现在 SQL 语句凡值是合法的几乎所有地方。但是 SQL 的 CASE 表达式与编程语言中的 CASE 语句不同，不能独立成句。

技巧 76 “理解 CASE 表达式”已经向读者展示了如何在 SELECT 语句中使用 CASE 表达式根据输入表之一中的列值改变结果表中的值。正如所了解的，“结果表”并不是磁盘上的物理表，而是一种可正确地模拟 SELECT 语句返回的查询结果以及 DBMS 可将查询结果用作别的 SQL 语句的“输入”或是“目标”表的概念性表。

除了改变虚拟的结果表之外，如果 CASE 表达式出现在 UPDATE 语句（而不是 SELECT 语句）中，可修改存储在物理表中的数据。例如，如果想要更新雇员记录中的 BONUS_LEVEL 列（与技巧 76 中显示的虚拟结果表中显示的值对照），可使用以下带有 CASE 表达式的 UPDATE 语句：

```
UPDATE employees
SET bonus_level =
    CASE WHEN total_sales < 10000 THEN 'Rookie'
        WHEN (total_sales >= 10000) AND
            (total_sales < 100000) THEN 'Associate'
        WHEN (total_sales >= 100000) AND
            (total_sales < 1000000) THEN 'Manager'
        WHEN (total_sales >= 1000000) THEN 'Vice President'
    END
```

当处理 CASE 表达式时，DBMS 取出满足 SQL 语句搜索条件行中的列值并将其用于确定 CASE 表达式中的第一个条件是否为 TRUE。如果是，CASE 表达式接受第一个 THEN 部分的值。如果不是 TRUE，DBMS 对 CASE 表达式中的第二个 WHEN（搜索）条件求值。如果为 TRUE，CASE 表达式接受第二个 THEN 部分的值。如果不是 TRUE，DBMS 测试第三个搜索条件，依此类推。

如果没有 CASE 表达式中的搜索条件求值为 TRUE，CASE 表达式接受 CASE 表达式中可选的 ELSE 子句中给定的值。如果没有搜索条件求值为 TRUE，则没有 ELSE 子句的 CASE 表达式返回 NULL。

当用于 UPDATE 语句中时，由 CASE 表达式（并接着用于更新表列）返回的值必须与被更新的表列具有相同的数据类型。因而，如果在 CASE 表达式的 THEN 子句中有实际的（字符串），且又试图更新数值类型的列，DBMS 将返回一条错误。

除了使用实际的字符串之外，可将列名和数学表达式用于计算由 CASE 表达式返回的值中。例如，只要 EMPLOYEES 表中的 BONUS_LEVEL 列有正确的值，就可使用以下 UPDATE 语句：

```
UPDATE employees
SET commission =
    CASE bonus_level
        WHEN 'Rookie' THEN total_sales * .01
        WHEN 'Associate' THEN total_sales * .05
        WHEN 'Manager' THEN total_sales * .15
        WHEN 'Vice President' THEN total_sales * .25
    END
```

根据与雇员的 BONUS_LEVEL 和 TOTAL_SALES 列相联系的佣金水平来设置 COMMISSION 列的值。

技巧 78 使用 CASE 表达式避免错误条件

除了修改结果表值和更新物理表中的数据值之外，还可使用 CASE 表达式来避免数学上是非法的或是侵犯了数据类型范围约束的计算。

例如，如果想要显示销售人员打电话而产生的销售的次数百分数，可使用以下 SELECT 语句：

```
SELECT id, name, (sales_calls / sales_count) * 100.00
       AS closing_percentage
FROM employees
WHERE department = 1
```

但是，如果 SALES_COUNT 的值为 NULL 或 0，DBMS 将产生一个错误（并停止语句的执行，因为除以 0 是非法的数学运算）。

通过向 SELECT 语句中添加 CASE 表达式可避免被 0 或 NULL 除的数学例外。例如，以下 SELECT 语句可使防止系统试图将 SALES_COUNT 列被 0 或 NULL 值除：

```
SELECT id, name, CASE WHEN sales_count > 0 THEN
                        sales_calls / sales_count * 100.00
                    ELSE 0
                END AS closing_percentage
FROM employees
WHERE department = 1
```

当 SALES_COUNT 大于 0 时，DBMS 执行 SALES_COUNT 除以 SALES_CALLS 的除法。作为对照，当 SALES_COUNT 为 0 或未知时，DBMS 将跳过除法并返回值 0。

有时数学上合法的计算也可引起侵犯了范围约束的结果。例如，假设使用以下 UPDATE 语句可从雇员的 GROSS_PAY 列中扣除健康保险的费用：

```
UPDATE payroll_records
SET net_pay = gross_pay - health_ins_deduction
```

如果 HEALTH_INS_DEDUCTION 大于 GROSS_PAY，则侵犯了用户定义的数据范围约束，支票上的 NET_PAY 数量不能小于 0！

为了避免写出负数的支票，可按下列例向 UPDATE 语句添加 CASE 表达式：


```
UPDATE payroll_records
SET net_pay = CASE WHEN gross_pay >= health_ins_deduction
    Then gross_pay - health_ins_deduction
    ELSE gross_pay
END
```

DBMS 仅当雇员所赚的钱至少为要扣除的量时才从 GROSS_PAY 中扣除 HEALTH_INS_DEDUCTION。

技巧 79 理解 NULLIF 表达式

NULLIF 函数是 ISNULL 的反函数。虽然 ISNULL 函数用非 NULL 值来代替 NULL，但 NULLIF 却是用 NULL 值来代替非 NULL 值。

例如，假设某人将 EMPLOYEES 表中的还没有指定定额的新雇员的 SALES_QUOTA 列设置为-1。在以下 SELECT 语句中的 NULLIF 表达式：

```
SELECT employee_id, first_name, last_name,
       NULLIF (SALES_QUOTA, -1) as Quota
FROM employees
```

通过将结果表中的 SALES_QUOTA 列中的-1 用 NULL 值来代替，从而在虚拟的结果表中更准确地显示销售定额。

NULLIF 表达式的句法如下：

```
NULLIF (<expression 1>, <expression 2>)
```

在对 NULLIF 表达式求值时，如果<expression 1>和<expression 2>有同样的值，DBMS 将返回 NULL 值。如果<expression 1>和<expression 2>有不同的值，DBMS 将返回<expression 1>值作为表达式的值。这样一来，在本例中，只要 SALES_QUOTA 列的值等于-1，NULLIF 表达式求值为 NULL。否则，NULLIF 表达式将返回当前行的 SALES_QUOTA 列的值。

NULLIF 中的任何一表达式（或两个表达式）都可是实际的值（数值的、日期时间或是字符常数）、数值、字符串日期或是 CASE 表达式或列名。但是，对<expression 1>和<expression 2>都使用实际的值没有什么实际意义。例如，以下 SELECT 语句：

```
SELECT
NULLIF ('match', 'match'), NULLIF ('no match', 'match')
```

生成的结果表如下：

```
-----
NULL, no match
```

这就告诉人们，当有两个相同的实际值的 NULLIF 表达式只能求值为 NULL，而两个不同的实际值中的 NULLIF 表达式求值为第一个实际值。

除了改变 SELECT 语句的虚拟结果表中的内容之外，还可在 UPDATE 语句中使用 NULLIF 表达式来改变实际（物理）表中的值。例如，假设关闭了销售办公室之一（office 6）。可使用以下 UPDATE 语句将以前指定给 office 6 的所有雇员的 OFFICE 列中放入 NULL 值：

```
UPDATE employees SET office = NULLIF (office, 6)
```

技巧 80 使用 COALESCE 表达式代替 NULL 值

COALESCE 表达式给出了一种用非 NULL 值来代替 NULL（或缺少的）数据的简单方式。

虽然可使用 CASE 表达式达到同样的目的，但以下 COALESCE 的句法比等价的 CASE 表达式更为紧凑并且或许更易于阅读：

```
COALESCE (<first expression>, <second expression>
         [...<last expression>])
```

DBMS 将 COALESCE 表达式的值设置为表达式列表中的第一个非 NULL 值。正如表达式的句法所表明的，COALESCE 表达式与 ISNULL 表达式（也用于用 NULL 值来代替所选的表达式）不同，在其表达式列表中可有两个以上的表达式。

为了计算 COALESCE 表达式的值，DBMS 以对列表中的第一个表达式求值开始。如果第一个表达式求值为非 NULL 结果，则 DBMS 返回其值作为 COALESCE 表达式的值。如果第一个表达式求值为 NULL，则 DBMS 对列表中的第二个表达式求值。如果第二个表达式求值为非 NULL 结果，则 DBMS 返回第二个表达式的值作为 COALESCE 表达式的值。如果第二个表达式也求值为 NULL，DBMS 继续对第三个表达式求值，依此类推。

最后，DBMS 从左向右读取，返回表达式列表中的第一个非 NULL 值作为 COALESCE 表达式的值。如果表达式列表中的所有表达式求值均为 NULL，COALESCE 表达式返回 NULL 值。

作为 DBMS 如何计算 COALESCE 表达式值的示例，假设想要列出销售代表的指定定额。另外还假设想要使用最小指定定额来作为雇员表中还未指定定额的任何销售代表的定额。为了产生清单，可执行以下 SELECT 语句：

```
SELECT employee_id, first_name, last_name,
       COALESCE (appt_quota,
                (SELECT MIN(appt_quota) FROM employees), 0) AS quota
FROM employees
WHERE department = 'Marketing'
```

在对 COALESCE 表达式求值之后，如果 APPT_QUOTA 列的值不是 NULL，DBMS 显示 APPT_QUOTA 列中的值。如果 APPT_QUOTA 为 NULL，DBMS 对 APPT_QUOTA 列计算 MIN 总计函数并显示最小指定定额作为雇员的定额，只要至少有一个销售代表有定额即可。如果所有的定额都是 NULL，MIN 总计函数返回 NULL，DBMS 将显示表达式列表中的最后的值 0，作为雇员的指定定额。

技巧 81 使用 COUNT(*) 总计函数对表中的行数计数

COUNT(*) 是一种总计函数，返回满足 SELECT 语句的 WHERE 子句中的搜索条件的行数。因而以下 SELECT 语句将显示 EMPLOYEES 表的行数：

```
SELECT COUNT(*) FROM employees
```

类似地，以下 UPDATE 语句：

```
UPDATE managers SET employees_managed =
       (SELECT COUNT(*) FROM employees
        WHERE manager = managers.employee_id)
```

将把 MANAGERS 表中的 EMPLOYEES_MANAGED 列设置为 EMPLOYEES 表中的 MANAGER 等于 MANAGERS 表中的 EMPLOYEE_ID 列的行数。

这样一来，本例中的 UPDATE 语句告诉 DBMS 将 MANAGERS 表中的每行中的

EMPLOYEES_MANAGED 列设置为该经理管理的雇员的计数。

注意：如果 COUNT(*) 的 SELECT 语句中没有 WHERE 子句，那么表中所有行都满足所谓的“搜索条件”。由此，在第一个例子中的 SELECT 语句将返回表中所有行的计数，因为表中的每行都满足 SELECT 语句忽略了的搜索条件。

如果 DBMS 在其系统表中存储表行数，COUNT(*) 将很快地返回甚至非常大的表行计数，因为 DBMS 可直接从系统表中提取行计数（不必从头到尾读取表并对物理表中的行计数）。在那些不在系统表中维护行计数的系统上，通过将索引的 NOT NULL 约束的列作为参数使用 COUNT()，则可能更快地对表行计数。

当使用 COUNT() 函数而不是 COUNT(*) 函数对表中的行计数时，请记住，仅当传递到 COUNT() 的列在任何行上都没有 NULL 值时，由 COUNT() 返回的值才与执行 COUNT(*) 函数的结果是等价的。

例如，以下 SELECT 语句中的 COUNT() 函数：

```
SELECT COUNT (employee_id) FROM employees
```

仅当 EMPLOYEES 表中没有行在 EMPLOYEE_ID 列有 NULL 值时，才与以下使用 COUNT(*) 函数的 SELECT 语句具有相同的值：

```
SELECT COUNT(*) FROM employees
```

由 COUNT() 函数返回的值可描述为传递到此函数的列中的非零值的计数。另一方面，COUNT(*) 可最准确地定义为返回表中的行值的总计函数。因而，由于 COUNT() 仅当作为参数传递的列中没有 NULL 值时才返回表中正确的行计数，所以仅当列参数受 NOT NULL 约束的限制时才可用 COUNT() 代替 COUNT(*)。

技巧 82 使用 COUNT(*) 总计函数对列中的数据值数计数

虽然使用 COUNT(*) 函数的目的是对表中的行数计数，但 COUNT() 函数可用于对一列中的数据值计数。与忽略了所有列值的 COUNT(*) 函数不同，COUNT() 函数检查一列（或多列）中的值并仅对那些值不是 NULL 的行计数。

例如，在 EMPLOYEES 表中给出以下数据值：

employee_id	first_name	last_name	quota	manager
1	Lancer	Michael	5	NULL
2	Michael	Lancer	5	1
3	William	Silverman	NULL	2
4	Walt	Wellinski	8	1
5	William	Silverman	8	2
6	NULL	Gomez	10	2
7	Walt	Frazier	10	NULL

以下 SELECT 语句：

```
SELECT COUNT(*) AS Row_Count,
       COUNT(last_name) AS Last_Name_Count,
       COUNT(manager) AS Manager_Count
FROM employees
```

将返回下面的结果表：

```
Row_Count Last_Name_Count Manager_Count
```

```
-----
```

```
7          7          5
```

这样一来，在本例中，COUNT(*)函数返回 EMPLOYEES 表中的行数，而不管任何表列中的值。相反，两个 COUNT()函数返回作为参数传递到每个函数中的列（LAST_NAME 和 MANAGER）中带有非 NULL 值的行数。

由于 LAST_NAME 列没有 NULL 值，COUNT(last_name)函数返回与 COUNT(*)函数相同的值。作为对照，在 MANAGER 列中有两个表行有 NULL 值，这就使得 COUNT(manager)函数返回值为 5，这比总数或是由 COUNT(*)函数报告的 EMPLOYEES 表中的行数小 2。

COUNT()函数的句法如下：

```
COUNT([ALL | DISTINCT] <expression>)
```

由于当既没有指定 ALL 也没有指定 DISTINCT 时，DBMS 默认地取 ALL，所以在使用 COUNT()函数时，通常忽略 ALL 限定词。技巧 83 “使用 COUNT(*)总计函数对列中惟一的和重复值计数”将向读者展示如何在 COUNT()函数中使用 DISTINCT 限定词对列中的惟一数据值计数。

要理解的重要事情是，COUNT()函数将接受列名、实际的字符串、数值或是将列内容组合成单一值的表达式。因而，在本例中，可使用以下 SELECT 语句对 FIRST_NAME 和 LAST_NAME 列均为 NULL 的行计数。

```
SELECT COUNT (first_name+last_name) FROM employees
```

注意：如果 DBMS 的具体实现对字符串串接使用双竖线运算符而不是加号 (+) 运算符，可把查询改写如下：

```
SELECT COUNT(first_name || last_name) FROM employees
```

如果想要只根据满足搜索条件的行对非 NULL 数据值计数，可向 SELECT 语句添加 WHERE 子句。例如，如果只想了解在 MANAGER 列为 2 的表行中的 FIRST_NAME 列中的非 NULL 值的数目，可使用以下 SELECT 语句：

```
SELECT COUNT(first_name) FROM employees WHERE manager = 2
```

注意：COUNT()函数的句法允许传递实际的字符串或是数字（即字符串或数字常数）作为其<expression>。使用实际的值（相对于列名或涉及列的表达式）总是使 COUNT()函数返回表的行数。毕竟，在 DBMS 对表的每一行对函数求值时，非 NULL 常数值绝不会是 NULL。因而 DBMS 将每一行都作为有非 NULL 值的受测试的列（常数值）。这样一来，例如，执行以下 SELECT 语句：

```
SELECT COUNT('constant value') FROM employees
```

将总是返回 EMPLOYEES 表的行数，因为实际的字符串“constant value”对表的每行来说都是非 NULL 值。

技巧 83 使用 COUNT(*)总计函数对列中的惟一和重复值计数

如果在作为参数传递到 COUNT()函数中的表达式之前插入 DISTINCT 约束，则函数将对一列中的惟一的、非 NULL 数据值计数。例如，给出下面的 EMPLOYEES 表中的数据值：

```
employee_ID first_name last_name  quota manager
```

```
-----
```

1	Lancer	Michael	5	NULL
2	Michael	Lancer	5	1
3	William	Silverman	NULL	2
4	Walt	Wellinski	8	1
5	William	Silverman	8	2
6	NULL	Gomez	10	2
7	Walt	Frazier	10	NULL

以下 SELECT 语句:

```
SELECT COUNT(first_name) AS Total_First_Names,
       COUNT(DISTINCT first_name) AS Unique_First_Names
FROM employees
```

将返回下面的结果表:

Total_First_Names	Unique_First_Names
6	4

注意: 显示表中的唯一行的一种简单方法是, 将表的 PRIMARY KEY 传递给 COUNT() 函数, 因为表的 PRIMARY KEY 列在表的每一行中都包括唯一的非 NULL 值。

通过使用字符串串接函数把值列组合成单一的复合的值(然后再作为参数传递给 COUNT() 函数)。从而可对跨多列的惟一值计数。

例如, 假设想要对 EMPLOYEES 表中的唯一的雇员姓和名计数。执行前面的 SELECT 语句会产生不准确的结果, 因为该语句中的 COUNT() 函数只检查 FIRST_NAME 列的惟一值。因而 DBMS 将第二个 WALT 作为重复的名字不加计数, 虽然全名 Walt Wellinski 与 Walt Frazier 并不相同。

为了准确地对唯一的雇员姓和名计数, 需要将 FIRST_NAME 和 LAST_NAME 列的内容组合成单一字符串, 然后将此字符串作为参数传递给 COUNT() 函数。这样做之后, 以下 SELECT 语句:

```
SELECT COUNT(first_name) AS Total_First_Names,
       COUNT(DISTINCT first_name) AS Unique_First_Names,
       COUNT(DISTINCT first_name + last_name) AS
       Unique_Full_Names
FROM employees
```

将返回下面的结果表:

Total_First_Names	Unique_First_Names	Unique_Full_Names
6	4	5

注意: 如果被串接列中的任何一列有 NULL 值, 串接的结果为 NULL, 则该行不会被 COUNT() 函数计数。在本例中, FIRST_NAME 列为 NULL, 而 LAST_NAME 列为 Gomez 的行未被 COUNT() 函数计数, 因为作为参数传递到 COUNT() 函数的表达式的结果(串接了名和姓的字符串)是 NULL。请记住, COUNT() 函数只对那些传递到函数中的参数不是 NULL 的行计数。

SQL 没有对列中的不惟一的(或重复的)值计数的特殊函数。但是, 可通过从总的列值计数中减去唯一列值计数而计算重复的列值数, 如以下 SELECT 语句所示:

```
SELECT COUNT(first_name + last_name) -
COUNT(DISTINCT first_name + last_name)
AS Dup_Name_Count
FROM employees
```

如果想要在重复列数据计数中包括 NULL 值的计数，可在上面的 SELECT 语句中用 COUNT(*) 函数代替第一个 COUNT() 函数。例如，以下 SELECT 语句：

```
SELECT COUNT(*) - COUNT(DISTINCT first_name + last_name)
AS Dup_Name_And_NULL_Count
FROM employees
```

将显示复合的 FULL_NAME (FIRST_NAME + LAST_NAME) 列中的重复和 NULL 值的计数。

技巧 84 使用 MS-SQL Server 的 CUBE 和 ROLLUP 运算符总计表的数据

SQL 的 GROUP BY 子句允许根据一个或多个列总计表中的数据。MS-SQL Server 提供两个运算符：CUBE 和 ROLLUP，可加强 GROUP BY 子句的总计能力。通过向 SELECT 语句添加 WITH ROLLUP 子句（或 WITH CUBE 子句），告诉 MS-SQL Server 对于列在语句的 GROUP BY 子句中的列生成附加的分类汇总和汇总。

例如，假设在 INVOICES 表中有下面的数据：

inv_date	inv_no	cust_id	product_code	qty
2000-01-01	1	1	1	1
2000-01-01	1	1	6	1
2000-01-01	1	1	3	1
2000-01-01	1	1	5	6
2000-03-01	2	9	1	5
2000-03-01	2	9	2	4
2000-02-01	3	7	2	4
2000-05-01	4	7	5	1
2000-05-01	4	7	4	3
2000-05-01	4	7	2	8
2000-01-01	5	4	5	3
2000-01-01	5	4	6	3
2000-06-01	6	1	5	4
2000-06-01	7	5	5	4

为了生成由顾客 4 和顾客 5 购买的产品的总结报告，可使用以下带 GROUP BY 子句的 SELECT 语句：

```
SELECT cust_id, product_code, SUM(qty) AS quantity
FROM invoices WHERE cust_id IN (4, 5)
GROUP BY cust_id, product_code
ORDER BY cust_id
```

对于当前示例数据，将产生下面的结果表：

cust_id	product_code	quantity
4	5	3
4	6	3

5 5 4

这样一来, GROUP BY 子句告诉 DBMS 将在 SELECT 清单中的非组合数据项“组合”(或总计)为在 GROUP BY 子句中列出的“类别”(或列)。

在本例中, QTY 列的和是惟一的非组合数据项。因而, GROUP BY 子句告诉 DBMS 为 INVOICES 表中的惟一的 CUST_ID 和 PRODUCT_CODE 组合计算 QTY 列值的总和。WHERE 子句告诉 DBMS 只显示顾客 4 和顾客 5 的数据。

如果正在使用 MS-SQL Server, 可添加 WITH ROLLUP 子句告诉 DBMS 显示每个顾客购买的总产品数(不管产品代码如何)和所有顾客购买的所有产品总数。对于本例中的数据来说, 以下 SELECT 语句 (WITH ROLLUP):

```
SELECT cust_id, product_code, sum(qty) AS quantity
FROM invoices WHERE cust_id IN (4, 5)
GROUP BY cust_id, product_code
WITH ROLLUP
ORDER BY cust_id
```

将生成下面的结果表:

cust_id	product_code	quantity
NULL	NULL	10
4	5	3
4	6	3
4	NULL	6
5	5	4
5	NULL	4

带有 NULL 值的每个附加的结果表行是由 ROLLUP 运算符生成的分类汇总。NULL 表明要进行分类汇总的列——或者换句话说, 列中的 NULL 值意味着对该列的“所有值”。例如, 第一行在 CUST_ID 和 PRODUCT_CODE 列都有 NULL 值, 这表明 QUANTITY 列包括由所有顾客购买的所有产品代码的总数。类似地, 结果表中的第四行只在 PRODUCT_CODE 有 NULL 值。因而第 4 行上的 QUANTITY 列显示由顾客 4 所购买的所有产品代码的总数。

请注意, 带 WITH ROLLUP 的 SELECT 语句的结果表只有一行在 CUST_ID 列上有 NULL 值。ROLLUP 运算符根据列在子句中的其余列对 GROUP BY 子句中的第一列将分类汇总累计成单一的总汇总, 这代表对于 GROUP BY 清单中的所有列的所有值在非组合列中的数量。

如果想要显示对 GROUP BY 子句中的第一列中那些对于列在子句的其余列的每一个惟一的列值组合的分类汇总, 可使用 WITH CUBE 子句代替 WITH ROLLUP 子句。对于本例来说, 以下 SELECT 语句:

```
SELECT cust_id, product_code, sum(qty) AS quantity
FROM invoices WHERE cust_id IN (4, 5)
GROUP BY cust_id, product_code
WITH CUBE
ORDER BY cust_id
```

将生成下面的结果表:

cust_id	product_code	quantity
4	5	3
4	6	3
4	NULL	6
5	5	4
5	6	3
5	NULL	7
NULL	5	7
NULL	6	6
NULL	NULL	13

NULL	NULL	10
NULL	5	7
NULL	6	3
4	5	3
4	6	3
4	NULL	6
5	5	4
5	NULL	4

这就包括了两个额外的行（行 2 和行 3），其中显示的是由所有顾客所购买的所有产品 5 和产品 6 的总数。

注意：在本技巧中的示例使用的 SELECT 语句的 GROUP BY 子句只列出了两列。这是为了减少结果表的容量而这样做的。MS-SQL Server 的 CUBE 和 ROLLUP 运算符将接受带多达 10 列清单的 GROUP BY 子句。

技巧 85 使用 MAX() 总计函数找出列中的最大值

当需要了解一列中的最大值时，可使用 MAX() 总计（或集合）函数。传递到 MAX() 函数中的列可以是数值、字符串或是日期时间数据类型。因此，如果想要显示顾客 1 最近的发票的日期时间，可使用以下 SELECT 语句：

```
SELECT MAX(inv_date) AS 'Date Last Inv for Cust 1'
FROM invoices WHERE cust_id = 1
```

产生下面的结果表：

```
Date Last Inv for Cust 1
-----
2000-06-01 00:00:00.000
```

类似地，如果想要显示 PRODUCTS 表中最高的 ITEM_COST，可使用以下 SELECT 语句：

```
SELECT MAX(item_cost) AS 'Max Item Cost' FROM products
```

来产生下面的 RESULTS 表：

```
MAX Item Cost
-----
1844.5100
```

MAX() 函数将返回与被传递的列同一数据类型的单一值。这样一来，在第一个例子中，MAX() 函数扫描 INV_DATE 列（为日期时间类型）的值并返回在该列找到的最大的日期时间值。类似地，第二个例子中的 MAX() 函数扫描数值列并返回 ITEM_COST 列中的最高的 MONEY 数据类型值。

注意：在确定列中的最大值时，MAX() 函数忽略 NULL 值。但是，如果在该列中的所有行都有 NULL 值，对该列 MAX() 函数将返回 NULL 值。

当需要了解一列中的最小值时，可使用 MIN() 总计（或集合）函数。

技巧 86 使用 SUM() 总计函数计算列值的总和

SUM() 总计（集合）函数返回一列中数据值的总和。由于 SUM() 将行中一列的值加在一起，传递到 SUM() 函数的列必须是数值数据类型。

例如，为了显示 EMPLOYEES 表中的所有雇员的总的销售佣金，可使用以下 SELECT 语句：

```
SELECT SUM(sales_commission) AS 'Total Commissions'  
FROM employees
```

来产生下面的结果表：

```
Total Commissions  
-----  
1072105.7900
```

虽然 MIN() 和 MAX() 函数每个都返回与传递到函数中的列的数据类型完全一样的值，但 SUM() 函数返回的数值类型既可与传递的列的数据类型相同，也可是比传递数据类型高的数据类型。例如，为了防止“溢出”错误，调用时对 SMALLINT 列的数值相加，SUM() 函数可能返回 INTEGER 类型——如果生成的总和大于 32,767 的话。

注意：SUM() 函数在对列中数值相加时忽略 NULL 值。但是，如果列中的所有值均为 NULL，则 SUM() 函数返回 NULL 作为其结果。

SUM() 函数的句法如下：

```
SUM([DISTINCT] <expression>)
```

由此，可让 SUM() 函数计算一列中所有惟一值的总和。例如，以下 SELECT 语句中的 SUM() 函数：

```
SELECT SUM(DISTINCT quantity_on_hand) FROM products
```

如果 QUANTITY_ON_HAND 列的值为 1、5、9、12、9 和 5 时返回 27。相反，在以下语句中的 SUM 函数：

```
SELECT SUM(quantity_on_hand) FROM products
```

对同样的列值将产生 41。

技巧 87 使用 AVG() 总计函数计算列中值的平均值

AVG() 总计（集合）函数返回一列中数据值的平均值。由于 AVG() 函数将一列中的值加起来再将和除以非 NULL 值的数目，所以被平均的列必须是数值数据类型。

为了显示 PRODUCTS 表中数据项（ITEM_COST）的平均价格，可使用以下 SELECT 语句：

```
SELECT AVG(item_cost) FROM products
```

与 SUM() 函数类似，AVG() 函数不一定返回与传递到函数的列完全相同的数据类型。例如，如果使用 AVG() 函数来确定 EMPLOYEE 表中的雇员的平均年龄，如以下语句所示：

```
SELECT AVG(age) FROM employees
```

AVG() 函数可返回浮点数值，虽然 AGE 列是整数类型（INTEGER 或 SMALLINT）。

注意：请检查一下具体的 DBMS 的系统文档，从而确定出现的转换之后的数据类型。例如，MS-SQL Server 会把对 INTEGER 执行 AVG() 函数的结果进行四舍五入，虽然其他 DBMS 实现会返回计算的平均值为未舍入的浮点数。

在计算平均值时，AVG() 函数忽略 NULL 值。因此，如果对 PRODUCT 表中的任何产品的 SALES_PRICE 列还没有设定，结果就被设置为 NULL，以下语句：

```
SELECT SUM(sales_price) / COUNT(*) FROM products
```

与下面语句不等价：

```
SELECT AVG(sales_price) FROM products
```

虽然 SUM()函数忽略 NULL 值，但 COUNT(*)函数却不忽略。

与 SUM()函数类似，AVG()函数的句法如下：

```
AVG([DISTINCT] <expression>)
```

可用来只计算列中那些惟一（或是不同的）值的平均。由此，如果 EMPLOYEES 表的 AGE 列包括值为 26、55、34、37、34 和 55，以下 SELECT 语句（计算惟一值的平均）：

```
SELECT AVG(DISTINCT age) FROM employees
```

将显示平均值为 38，虽然下面的 SELECT 语句执行的平均值为 40：

```
SELECT AVG(age) FROM employees
```

注意：正如前面所提到的，AVG()函数在计算一列的平均值时忽略 NULL 值。但是，如果所有行在该列上都有 NULL 值，则 AVG()函数将为该列返回 NULL。

技巧 88 使用带 AVG()函数的 WHERE 子句确定表中所选行的平均值

正如读者在技巧 87“使用 AVG()总计函数计算列中值的平均值”中所学习的，AVG()总计（集合）函数返回一表列中的数据值的平均值。如果不想对列中的所有值求平均，则可在 WHERE 子句中使用搜索条件来限制 DBMS 将包括在总计函数的计算中的行（因而也就是数据值）。

例如，为了显示公司中所有销售人员的毛销售额的平均值，可使用以下 SELECT 语句：

```
SELECT AVG(gross_sales) AS 'Avg Gross Sales' FROM salesreps
```

为了只计算那些分配给 Nevada 的所有销售人员的毛销售额（与公司范围内的所有销售人员对比），就要将传递到 AVG()函数中的 GROSS_SALES 数据值限制为只有 Nevada 的销售人员，可向 SELECT 语句加入 WHERE 子句，如下语句所示：

```
SELECT AVG(gross_sales) AS 'Avg NV Gross Sales'
FROM salesreps
WHERE sales_territory = 'NV'
```

当执行 SELECT 语句时，DBMS 对表中的每行都对 WHERE 子句中的搜索条件求值。只有来自那些搜索条件求值为 TRUE 的行中的数据值才传递到总计函数中（默认情况下，如果 SELECT 语句中没有 WHERE 子句，表中的每一行都满足搜索条件）。

在本例中，只有来自 SALES_TERRITORY 等于 NV（Nevada 的缩写）的行中的 GROSS_SALES 才包括在由 AVG()函数进行的平均值计算中，在结果表中 Avg NV Gross Sales 的标题之下显示结果表。

除了显示表中某些或所有行的列的平均值，可使用 AVG()函数作为 SELECT 语句的 WHERE 子句中搜索条件的一部分。例如，假设想要显示那些 GROSS_SALES 大于所有雇员的平均值的销售人员的平均 QUOTA 和 GROSS_SALES 列值。以下 SELECT 语句：

```
SELECT AVG(quota) AS 'Avg Quota for Above Avg Sales',
       AVG(gross_sales) AS 'Avg Above Avg Sales'
FROM salesreps
WHERE gross_sales > (SELECT AVG(gross_sales)
FROM salesreps)
```

将确定所有销售人员的平均 GROSS_SALES 值，然后将每个销售人员 GROSS_SALES 列对照平均值加以比较。只有那些销售人员的 GROSS_SALES 值大于总体平均的 GROSS_SALES 的行才会传递到 AVG(quota)和 AVG(gross_sales)函数中。

技巧 89 理解 SELECT 语句中的总计函数如何产生单表结果

在技巧 81~技巧 87 中,读者已经学习了有关 SQL 的总计函数 COUNT()、MAX()、MIN()、SUM()和 AVG()的知识。虽然每个函数执行不同的功能,但所有函数都有将来自多个表行的列值总计(总结)为单一值的特性。由总计函数返回的值接着就可用在搜索条件或表达式中,或是作为 SELECT 语句的一系列显示出来。

当执行一条 SELECT 语句来显示由一个或多个总计函数计算的值时,DBMS 执行以下步骤:

1. 生成一个虚拟的中间表,用来代表 SELECT 语句的 FROM 子句中的表(或 CROSS JOIN)。
2. 如果有一条 WHERE 子句,就对中间表的每一行对其搜索条件求值。清除那些 WHERE 子句求值为 FALSE 或 NULL(未知的)的行——也就是说,只保留那些搜索条件求值为 TRUE 的行。
3. 使用更新了的中间表中的值来计算 SELECT 语句的选择子句中的总计函数的值。
4. 作为单个结果表中的列值显示由每个总计函数计算得来的值。

或许解释 DBMS 为总结查询生成结果的方式的最好方法是把查询的执行看作是有两个不同的阶段。在第一阶段中,DBMS 执行详细查询处理中的步骤(此内容读者已在技巧 62“理解处理 SQL 的 SELECT 语句所涉及的步骤”学习过)。生成的多行、多列的中间输入表中包括 SELECT 语句中的 FROM 子句中的所有表中的所有列,而且满足 WHERE 子句中的搜索标准的所有行。在第二阶段,DBMS 在查询的选择子句中使用总计函数将多行中间输入表总结为单一值,然后作为单行的结果表中的列显示出来。

如果把选择子句中的总计函数看作是“指导”DBMS 总结数据以产生结果的单行,则可以理解为什么以下 SELECT 语句是非法的:

```
SELECT dept, COUNT(*) FROM employees
```

毕竟,列引用 DEPT 告诉 DBMS 提供一个多行的清单,即显示 EMPLOYEES 表中每个雇员的部门号的虚拟表行。遗憾的是,这直接与选择清单中的第二项的指示相冲突。总计函数 COUNT(*)告诉 DBMS 提供单行的结果表(其中有显示输入表中的行数的一列)。

这样一来,在选择子句中列出列和总计函数的 SELECT 语句是非法的,因为列出的列告诉 DBMS 执行一个详细的查询,而同时总计函数告诉 DBMS 执行总结查询。

注意:如果列在选择子句中的所有列还出现在 SELECT 语句的 GROUP BY 子句中时,在选择子句中混合使用列清单和总计函数是非法的。因而以下 SELECT 语句是非法的:

```
SELECT dept, COUNT(*) FROM employees GROUP BY dept
```

(读者将在技巧 200“使用 GROUP BY 子句根据单一列值组合行”和技巧 201“使用 GROUP BY 子句根据多列组合行”中学习单列和多列的 GROUP BY 子句如何将详细查询转化为总结查询。)

技巧 90 使用 AND 逻辑连接符对表行进行多条件选择

SELECT 语句中的 WHERE 子句允许指定在选择要在结果表中显示的数据时 DBMS 使用的搜索条件。虽然 WHERE 子句是可选的,但几乎每条 SELECT 语句都有 WHERE 子句。毕竟,没有 WHERE 子句,结果表将包括输入表中的每行数据。由于数据库表常常会增长到几

百万行,显示整个表的内容既不实际也没有什么用——特别是因为对最不具学术价值的查询的答案可能会位于一部分的表数据的总结上。

例如,假设想要当前工作于市场部门的所有雇员的清单。可使用以下查询:

```
SELECT emp_id, first_name, last_name FROM employees
WHERE dept = 'Marketing'
```

如果没有 WHERE 子句, SELECT 语句将给出列出公司中所有雇员的结果表。如果只想把营销部门的工作人员指定为基层主管,则“多余的”非营销雇员没有什么用。

即使有 WHERE 子句,示例的 SELECT 语句也不能将雇员清单缩小得足够多。毕竟,想要管理的只是那些活跃的营销代表,而雇员表包括所有的营销代表——包括那些不再被公司雇用的代表(为了纳税的目的,大多数雇员表都至少要包括已经解雇的人员到本年底)。

由于 WHERE 子句中的单个搜索条件只能测试一个数据值,测试多个值(如 DEPT 和 EMPLOYMENT_STATUS)就需要多个搜索条件(或复合的)判式。这正是 AND 逻辑连接符有用武之地的地方。

AND 逻辑连接符允许指定额外的必须为真的搜索条件,以便 SELECT 语句根据 DBMS 的 IS 测试来显示数据值。对于本例来说,可使用 AND 逻辑连接符将两条搜索条件添加到 WHERE 子句中,如下所示:

```
SELECT emp_id, first_name, last_name FROM employees
WHERE dept = 'Marketing'
AND employment_status = 'Active'
```

现在 DBMS 将只对那些在 DEPT 列中的值为 Marketing,而且(AND)EMPLOYMENT_STATUS 列的值为 Active 的行返回 EMP_ID、FIRST_NAME 和 LAST_NAME。由此,就可得到所有当前为公司工作的营销人员的清单。

在 WHERE 子句可按需要使用任意多个 AND 连接符来指定输入表中的行必须满足的所有搜索条件,以便让其选择包括在结果表中的值。每个附加的 AND 连接符允许添加另一个搜索条件。这样一来,对本例来说,如果只想要那些当前被雇用的而且还未指定主管的营销人员的清单,可添加第二个 AND 连接符以包括第三个搜索条件:

```
SELECT emp_id, first_name, last_name FROM employees
WHERE Dept = 'Marketing'
AND employment_status = 'Active'
AND supervisor IS NULL
```

技巧 91 使用 OR 逻辑连接符对表行进行多条件选择

WHERE 子句中的每个搜索条件可对单列执行条件测试。这样一来,为了查询 STUDENTS 表得到所有主修化学的学生的清单,可使用以下 SELECT 语句:

```
SELECT student_id, first_name, last_name FROM students
WHERE major = 'Chemistry'
```

如果决定是否要在结果表中包括一行的数据涉及检查不止一列或以多种方式检查同一列,可向 WHERE 子句的每个 DBMS 要进行的测试添加附加的搜索条件。在技巧 90“使用 AND 逻辑连接符对表行进行多条件选择”中,读者已经学习了如何使用 AND 逻辑连接符在 WHERE 子句中将多个搜索条件连接在一起。

正如读者现在所了解的，AND 连接符告诉 DBMS，它所连接的所有搜索条件必须都为 TRUE，才能使 WHERE 子句求值为 TRUE。如果想让 DBMS 当 WHERE 子句中的一个或多个搜索条件为 TRUE 时包括来自当前行的数据，可使用 OR（而不是 AND）将搜索条件连接起来。

例如，假设想要得到主修化学、生物学或数学的学生清单。可提交以下 SELECT 语句来产生清单：

```
SELECT student_id, first_name, last_name, major
FROM students
WHERE major = 'Chemistry' OR major = 'Biology'
OR major = 'Mathematics'
```

在 DBMS 执行了 SELECT 语句之后，结果表将包括来自 MAJOR 列为 chemistry（化学）、biology（生物学）或 mathematics（数学）的学生 ID、姓名和主修信息，也就是说，如果由 OR 连接的任何一个搜索条件为 TRUE，WHERE 子句求值为 TRUE，DBMS 将在结果表中包括来自受检行中的数据。

第5章 理解 SQL 的事务处理和事务处理日志

技巧 92 理解 SQL 的事务处理过程

数据库的事务处理是为了完成指定任务的一个或多个都必须成功执行的语句序列。SQL-92 规定, DBMS 必须提供 BEGIN TRANSACTION 和 END TRANSACTION 用来将 DBMS 要自动执行的命令组合起来——既可执行组中(用 BEGIN TRANSACTION 和 END TRANSACTION 括起来的)所有语句, 也可以一条也不执行。

如果 DBMS 已经执行了事务处理中的某些语句, 而应用程序却异常中止, 或者有某些其他硬件和软件故障, DBMS 要负责取消已经执行的任务(语句)并将数据库表恢复到原始的未修改的状态。在技巧 95 “使用 ROLLBACK 语句取消对数据库对象所做的改变”中, 读者将学习如何使用 ROLLBACK 语句, 从而当改变了主意想要返回更新前的状态, 删除事务处理期间成功地执行了的语句时, 利用 DBMS 的“取消”修改的能力。

组合在一起形成事务处理的 SQL 语句(处于 BEGIN TRANSACTION 和 END TRANSACTION 语句之间)通常是独立的, 而且必须全部执行才能维持数据库的一致性。例如, 假设正在存货目录中改变产品的货品数目。为达到此目的, 必须做以下事项:

- 检查 PRODUCTS 表, 确保新的 PRODUCT_CODE 代码不属于已有的产品
- 更新 INVENTORY 表中的 PRODUCT_CODE, 使得某人用新的 PRODUCT_CODE 查询表时能够找出存货中的产品的数量
- 更新 ORDERS 表中的 PRODUCT_CODE, 使得仓库工作人员了解要从存货目录中提出哪种货品(根据产品代码)并将其发给顾客
- 更新 INVOICES 表中的 PRODUCT_CODE, 使得在顾客的票据上正确地打印出货品价格和描述
- 更新 PRODUCTS 表中的产品代码, 使得在使用新的主 PRODUCT_CODE 查询时, 货品清单显示出正确的产品描述和价格

如果事务处理中的任何更新没有完成, 数据库就变得不一致。例如, 如果 DBMS 不能改变 INVENTORY 表中的 PRODUCT_CODE 值, 则以下 SELECT 语句:

```
SELECT
    product_code, description, cost, vendor, serial_number
FROM inventory, products
WHERE inventory.product_code = products.product_code
```

将不再生成正确的结果, 因为 INVENTORY 表中的 PRODUCT_CODE 引用了在 PRODUCTS 表(其中存储有存货目录中每种产品的 DESCRIPTION、COST 和 VENDOR)中不存在的 PRODUCT_CODE。

从本例中可以看出, 事务处理中的每条语句执行部分任务, 例如在几个表之一中改变 PRODUCT_CODE 代码。另外, 所有任务必须完成才能成功地完成手边的整个任务, 如将

PRODUCT_CODE 在整个数据库中从一个值改变为另一个值。

有关事务处理过程的重要事情是, DBMS 将使用事务处理来保持数据库的一致性。由于 DBMS 将事务处理过程中的语句看作是工作的最小单位, 因而必须要么成功地执行所有语句, 要么通过把被事务处理中的语句修改的数据和数据库对象恢复到原始的未修改的状态, 使其看起来好像是什么语句也没有执行一样。

技巧 93“理解 ANSI/ISO 的事务处理模型”将解释大多数数据库产品如何自动地探测 SQL 事务处理过程的开始和结束, 技巧 96“理解 MS-SQL Server 的事务处理模型”将教授读者如何使用 BEGIN TRANSACTION 和 END TRANSACTION 语句将语句组合到 MS-SQL Server 的事务处理中。

技巧 93 理解 ANSI/ISO 的事务处理模型

在技巧 92“理解 SQL 的事务处理过程”中, 读者已经学习了, 事务处理是 DBMS 作为单个工作单元处理的 SQL 语句序列。换句话说, 事务处理中的语句被认为是最小的工作单元——要么所有语句都要成功执行, 要么取消所有已做的工作, 使得数据看起来好像是什么语句也没有执行一样。

ANSI/ISO 的事务处理模型定义了 COMMIT 和 ROLLBACK 语句的作用(这部分将在技巧 94 和 95 中加以讨论) 并指明, 事务处理自动地以用户或应用程序提交到 DBMS 的第一条语句开始。由此, 正在使用遵循 ANSI/ISO 的事务处理模型(例如, DB2)的用户总是在事务处理之内操作。

ANSI/ISO 的事务处理自动以一条 SQL 语句的执行开始并继续执行后续的语句直到以图 5.1 中所示的 4 种方式之一终止。

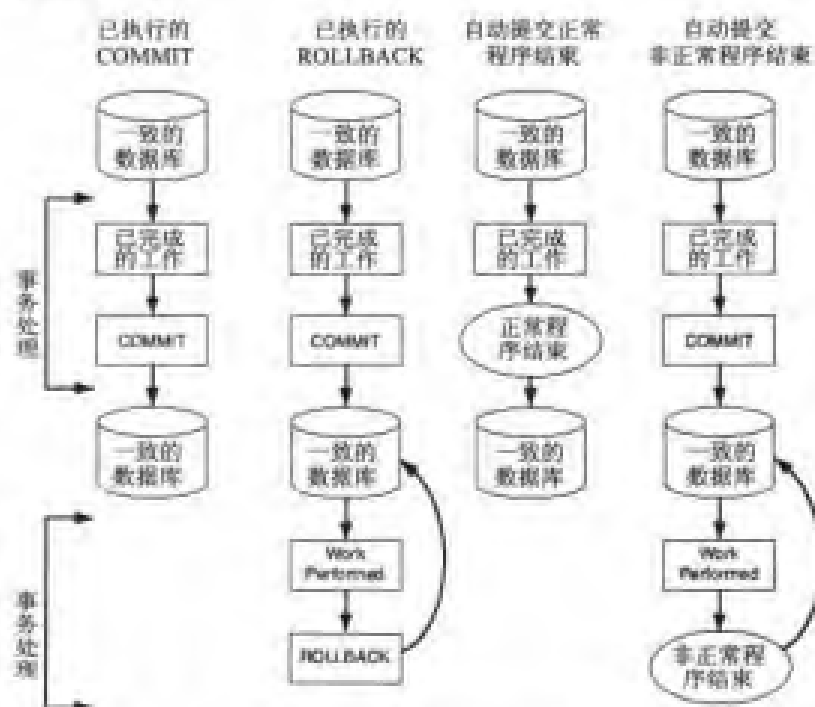


图 5.1 成功和不成功的事务处理之后的数据库状态

请注意, ANSI/ISO 的事务处理模型的定义特性是, 事务处理不必明确地开始。用户或应

用程序总处于事务处理之中，因为只要当前的事务处理以以下事件结束，则新的事务处理就开始了：

- 用户或应用程序执行一条 COMMIT 语句，表明事务处理中的所有语句都成功地执行了。COMMIT 语句使得数据库的改变成为永久的。
- 用户或应用程序执行了一条 ROLLBACK 语句，中断事务处理并删除事务处理期间所执行的任何工作（ROLLBACK 语句完成时，数据库对象和值都恢复到其原始的事务处理之前的状态，就好像没有执行过任何语句一样）。
- 应用程序正常终止。当应用程序正常终止时，DBMS 自动发出 COMMIT 语句，这就使得应用程序执行时所做的更新成为永久的。
- 应用程序异常中止。当应用程序异常中止时，DBMS 中断事务处理并执行 ROLLBACK。因此，如果应用程序异常中止，DBMS 取消任何程序对数据库的更新，将数据库恢复到中止的程序执行以前的状态。

注意：也可把硬件故障看作终止事务处理的第 5 种方式。如果硬件出了故障而 DBMS 重启之后，它将取消任何部分完成的事务处理，然后才能允许用户或应用程序访问数据库。简短地说，当 DBMS 在硬件故障（如断电）之后重启时，自动地发出 ROLLBACK 命令。

ANSI/ISO 的事务处理模型并不使用交互式的多语句的事务处理。事实上，大多数 DBMS 产品在默认情况下都将交互式会话变为自动提交模式，在这种模式下，当用户提交一条数据定义语言（DDL）或数据操作语言（DML）语句时，如以下语句：

```
UPDATE products SET sales_price = sales_price * 1.2
```

DBMS 自动地开始事务处理，然后只要 DBMS 成功地执行了其任务，就用一条 COMMIT 语句结束事务处理。由于只要 SQL 语句完成其执行，则 DBMS 就发出 COMMIT 命令，因而在自动提交模式下的交互会话期间，不能使用 ROLLBACK 取消由成功执行的语句所完成的工作。另外，只要提交了下一条语句用于处理，DBMS 就开始新的事务处理。因此，虽然当第二条语句不能执行时想要取消这两条（包括前一条）语句，但 DBMS 将只取消从失败的第二条语句中完成的工作（到第二条语句执行时，DBMS 已经使用一条 COMMIT 命令使第一条语句的工作成为永久的）。

技巧 94 和技巧 95 “使用 ROLLBACK 语句取消对数据库对象所做的改变”中将向读者展示如何在交互式会话中禁用自动提交模式，此时将能够执行多语句的事务处理并取消由一条或多条在交互式会话期间提交的语句所完成的工作。

现在要了解的重要事情是，ANSI/ISO 的事务处理模型（定义了 COMMIT 和 ROLLBACK 的作用）指明，一个 SQL 的事务处理自动以用户或应用程序执行的第一条语句开始。在技巧 96 “理解 MS-SQL Server 的事务处理模式”中，读者将学到，非 ANSI/ISO 事务处理模型在交互会话下的默认情况仍然是自动提交，但在程式化的 SQL 中，需要程序执行一条明确的 BEGIN TRANSACTION 语句来开始一项事务处理。

技巧 94 理解何时使用 COMMIT 语句

在 SQL 的事务处理期间，由语句所完成的工作，在永久地写入数据库之前都可以取消（使用 ROLLBACK 语句）。COMMIT 告诉 DBMS 使数据库的变化成为永久的。

正如读者在技巧 93 “理解 ANSI/ISO 的事务处理模型”中所学习的，大多数 DBMS 产品，

当接受交互式查询和更新时，都处理于默认的自动提交模式下。因此，当在 MS-SQL Server Query Analyzer（上部）的输入窗格中键入以下 UPDATE 语句，然后单击 Execute Query（执行查询）按钮：

```
UPDATE products SET sales_price = sales_price * 1.2
```

如图 5.2 所示，DBMS 将在 PRODUCTS 表中将所有产品的 SALES_PRICE 值增加 20%。

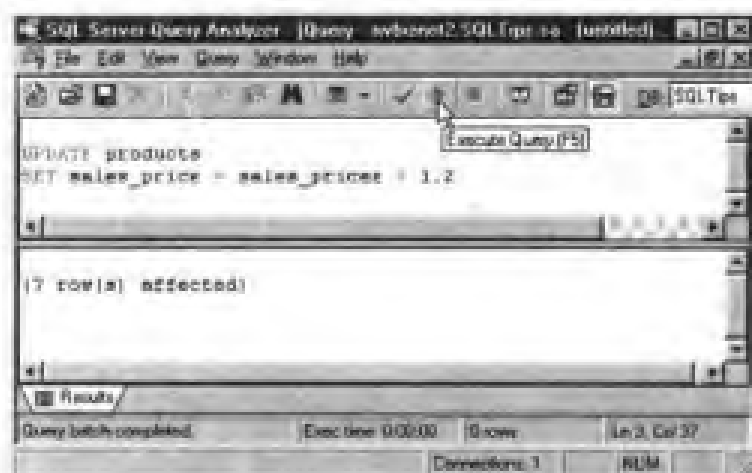


图 5.2 一个 MS-SQL Server's Query Analyzer 的交互会话

由于 Query Analyzer 提供与 MS-SQL Server 数据库（在本例中是 SQL Tips）的交互连接，DBMS 在 UPDATE 语句对数据库做出最后的改变并将控制权返回 Query Analyzer 之后，自动地发出 COMMIT 语句。这样一来，当工作于交互式的自动提交模式时，不需要执行 COMMIT 语句使工作永久地写入数据库。

与任何事务处理过程一样，在交互式会话期间，如果由向 DBMS 提交一条 SQL 语句开始的事务处理未运行到成功完成，则 DBMS 将取消所完成的工作。因而，在本例中，如果服务器在更新了 PRODUCTS 表中的半数产品的 SALES_PRICE 之后断电或是死锁，DBMS 将自动地将所有产品的 SALES_PRICE 设置回增加了 20% 之前的状态。

注意：明显地，在可以取消部分更新之前，服务器必须再次上线。在服务器重启了 DBMS 之后，DBMS 将检查其事务处理日志，从而找出任何部分地执行以及未提交的事务处理。在允许任何用户或应用程序与数据库连接之前，DBMS 将取消（ROLLBACK）由中止的和未提交的事务处理所完成的工作。

要了解的重要事情是，当 DBMS 执行一条 ROLLBACK 语句时，就取消最后一条 COMMIT 语句之后所有对数据库所执行的工作。当在交互模式下执行 SQL 语句时，ROLLBACK 只取消由最近的部分执行的语句所做出的改变（请记住，在交互会话期间的默认的自动提交模式下，完全执行的语句会自动地提交）。

在编程式的 SQL 中，执行一条明显的 COMMIT 语句变得重要，在此情况下，用户使用一种应用程序向 DBMS 发送 SQL 语句。在编程式的 SQL 会话期间，事务处理是以 BEGIN TRANSACTION 语句开始的。

注意：正如在技巧 93 中所学习的，遵循 ANSI/ISO 的事务处理模型的 DBMS，当应用程序提交其第一条 SQL 语句时，将自动地开始事务处理（不必有明显的 BEGIN TRANSACTION 语句）。

由应用程序提交的所有语句都变为事务处理的一部分——在应用程序提交一条 ROLLBACK 或 COMMIT 语句或是结束（既可是正常结束也可是异常结束）时，事务处理才会结束。这样一来，如果有一个应用程序，其中有如下的语句流：

```
BEGIN TRANSACTION
MOD 1...
MOD 2...
MOD 3...
MOD 4...
```

在应用程序正常结束之前，4 条修改并未提交（未成为永久的）。由此，异常结束将使得 DBMS 取消由应用程序体中的所有 4 条 SQL 语句所做的修改。如果想要让前两条修改仍然保留在数据库中，而不管修改 3 和 4 是否成功地执行，可让应用程序在提交 MOD 2 之后执行一条 COMMIT 语句：

```
BEGIN TRANSACTION
MOD 1...
MOD 2...
COMMIT TRANSACTION
BEGIN TRANSACTION
MOD 3...
MOD 4...
```

注意：如果正在使用遵循 ANSI/ISO 事务处理模型的 DBMS，程序在 MOD 1 之前不需要执行明显的 BEGIN TRANSACTION 语句。正如读者在技巧 93 中所学习的，随着 COMMIT 或 ROLLBACK 语句之后的第一条语句的执行，ANSI/ISO 事务处理模型的 DBMS 自动地开始一个新的事务处理。

现在要了解的重要事情是，COMMIT 语句表明成功地完成了 SQL 的事务处理并使得 DBMS 将事务处理中的语句所完成的工作永久化。提交的工作不能用 ROLLBACK 取消，而在正常或是异常关闭之后 DBMS 重新启动时也不能取消。

技巧 95 使用 ROLLBACK 语句取消对数据库对象所做的改变

在技巧 94 “理解何时使用 COMMIT 语句”中，读者学习了如何通过执行一条 COMMIT 语句，使作为 SQL 事务处理的一部分所完成的工作成为永久的，在永久性地写入数据库之前，可使用 ROLLBACK 语句取消由几乎所有 SQL 语句所做的改变，在 MS-SQL Server 中，其中包括使表和使用 DROP TABLE 语句从数据库中删除的数据回到原处！

在交互式会话期间，DBMS 默认为自动提交模式，这就意味着，DBMS 自动地提交（使之成为永久的）每一条成功执行的 SQL 语句的操作。因而，如果执行了以下语句：

```
DELETE FROM employees
```

就会从 EMPLOYEES 表删除所有行，不能使用以下语句使删除的数据再恢复：

```
ROLLBACK TRANSACTION
```

请记住，当 DBMS 接受了 DELETE 语句，然后当它从 EMPLOYEES 表删除最后一行时，自动地执行一条 COMMIT TRANSACTION 语句。因此，如果在执行了 DELETE 语句之后立即执行 ROLLBACK TRANSACTION，此时没有可取消的未提交的事务处理。

如果想能够取消在交互式会话期间由所执行的 SQL 语句所造成的变化，首先必须在交互

式会话中禁用自动提交模式。

例如, MS-SQL Server 允许通过执行以下语句禁用自动提交模式:

```
BEGIN TRANSACTION
```

注意: 必须检查一下系统文档, 从而找出自己的系统中用于禁用自动提交模式所用的特定语句。虽然 MS-SQL Server 使用 BEGIN TRANSACTION 语句, 但 DB2 使用 UPDATE COMMAND OPTIONS USING c OFF 语句。可在索引的 "transaction"、"commit" 或 "auto-commit" 项下查找。

在执行了 BEGIN TRANSACTION 语句之后, DBMS 将接下来的 SQL 语句看作是多条语句的事务处理的一部分。由此, 如果执行以下语句序列:

```
MOD 1  
BEGIN TRANSACTION  
MOD 2  
MOD 3  
MOD 4  
ROLLBACK TRANSACTION
```

DBMS 将取消由 BEGIN TRANSACTION 之后的 3 条 SQL 语句 (MOD 2、MOD 3、MOD 4) 所完成的工作。因而, 在执行了 ROLLBACK TRANSACTION 语句之后, 在本例中, 数据库看起来完全与执行了 SQL 语句 MOD 1 之后所处的状态相同。

注意: ROLLBACK TRANSACTION 并不取消由 MOD 1 所完成的工作, 因为 MOD 1 是在自动提交模式下执行的。结果, 在成功地完成由 MOD 1 代表的 SQL 语句时, 由 MOD 1 所引起的变化自动地提交 (成为永久的)。

令人困惑的是, 如果 DBMS 遵循 ANSI/ISO 的事务处理模型 (MS-SQL Server 和 Sybase 不遵循), 在交互式会话期间执行示例中的 SQL 语句将与使用编程式 SQL 通过应用程序向 DBMS 提交同样的语句序列有不同的影响。

在交互式会话期间, 不管是使用交互的还是编程的 SQL, 所执行的 ROLLBACK TRANSACTION 将具有同样的影响——数据库将只保留由 MOD 1 的代表的 SQL 语句所执行的改变。

如果示例语句是在应用程序中对遵循 ANSI/ISO 事务处理模型的 DBMS 所执行的, ROLLBACK TRANSACTION 将与由 MOD 2、MOD 3 和 MOD 4 所引起的改变一起也取消由 MOD 1 所引起的改变。但是, 如果示例中的语句是在应用程序中对不遵循 ANSI/ISO 事务处理模型的 DBMS 所执行的 (如 MS-SQL Server 和 Sybase), ROLLBACK 将不随 MOD 2、MOD 3 和 MOD 4 所引起的改变一起取消由 MOD 1 所引起的改变。

行为上的不同的原因是, ANSI/ISO 事务处理模型自动地以应用程序提交的第一条语句开始事务处理。这样一来, 当由对遵循 ANSI/ISO 事务处理模型的 DBMS 运行的应用程序提交 ROLLBACK TRANSACTION 时, MOD 1 是受 ROLLBACK TRANSACTION 影响的事务处理的一部分。

在使用 ROLLBACK 取消数据库的变化时, 要理解的重要事情是, ROLLBACK 取消在执行了最近的 COMMIT 语句之后所完成的所有工作。因而, 如果正在交互式会话期间之内执行 SQL 语句, 并想能够通过执行 ROLLBACK 语句取消所做工作, 或是想要执行多语句的事务处理, 一定要禁用默认的自动提交模式。

注意：请考虑使用明显的 BEGIN TRANSACTION 和 END TRANSACTION 语句块（只要可能的话），从而使 SQL 代码可在遵循 ANSI/ISO 事务处理模型（如 DB2）和那些不遵循此模型的（如 Sybase 和 MS-SQL Server）DBMS 之间是可移植的。

技巧 96 理解 MS-SQL Server 的事务处理模型

正如读者在技巧 92 “理解 SQL 的事务处理过程”和技巧 93 “理解 ANSI/ISO 的事务处理模型”中所学习的，SQL 的事务处理是一个语句序列，其中所有的语句都必须成功执行，才能完成一项任务。所有 DBMS 产品都完成由事务处理中的语句所指定的工作，或全部完成，或什么也没有发生——其中没有部分执行。要么是 DBMS 成功地执行了事务处理中的所有语句，要么是系统取消了所完成的工作并将数据库恢复到事务处理开始之前的状态。另外，所有 DBMS 产品都支持 ROLLBACK 语句，从而取消在事务处理中完成的，但还没有被 COMMIT 语句变为永久的工作。

MS-SQL Server 的事务处理模型类似于 ANSI/ISO 的事务处理模型的方面在于以下几条：

- 以事务处理作为最小单元执行事务处理中语句，要么所有都执行，要么什么也不执行。
- 结束事务处理并在接受了 COMMIT 语句时使所完成的工作成为永久的。
- 当程式的 SQL 应用程序正常结束时，自动地执行 COMMIT 语句。
- 允许用户执行 ROLLBACK 语句来结束事务处理并取消任何事务处理中完成的未提交的工作。
- 当编程的 SQL 应用程序异常结束时，自动地执行一条 ROLLBACK 语句。

但是，与 ANSI/ISO 的事务处理模型不同，MS-SQL Server 会：

- 不自动地随着程式的 SQL 应用程序的第一条语句开始事务处理。在把第一条语句看作是事务处理的一部分之前，MS-SQL Server 需要程序明确地通过执行 BEGIN TRANSACTION 语句将语句组合在事务处理内。
- 包括一条 SAVE TRANSACTION 语句，可用于建立事务处理中的存储点，使得用户可以取消某些事务处理语句（并非全部）所完成的工作。

MS-SQL Server 的存储点在带有大量语句时，特别有用之处在于，允许应用程序取消所做工作并恢复到事务处理中的特定点。由此，应用程序可对存储点使用 ROLLBACK 语句，重新做出事务处理中的部分语句所做的工作，而不必从头开始。例如，如果让 MS-SQL Server 执行以下语句：

```
BEGIN TRANSACTION
CREATE TABLE trans_table
  (row_number SMALLINT, description VARCHAR(35))
INSERT INTO trans_table VALUES (1,'Inserted Row 1')
INSERT INTO trans_table VALUES (2,'Inserted Row 2')
SAVE TRANSACTION savepoint_1

DELETE FROM trans_table WHERE row_number = 2
INSERT INTO trans_table VALUES (3,'Inserted Row 3')
INSERT INTO trans_table VALUES (4,'Inserted Row 4')
SAVE TRANSACTION savepoint_2
DELETE FROM trans_table WHERE row_number = 1
```

```
DELETE FROM trans_table WHERE row_number = 3
INSERT INTO trans_table VALUES (5, 'Inserted Row 5')
UPDATE trans_table SET description = 'All Rows Updated'
```

```
ROLLBACK TRANSACTION savepoint_2
```

```
UPDATE trans_table
  SET description = 'Row 1 After ROLLBACK to 2'
DELETE trans_table WHERE row_number = 4
```

```
COMMIT TRANSACTION
```

以下 SELECT 语句:

```
SELECT * FROM trans_table
```

的结果表如下:

row_number	description
1	Row 1 After ROLLBACK to 2
3	Inserted Row 3

很明显, 与示例不同, 实际的应用程序将使用某种决策机制, 如 Transact-SQL 语句 IF @@ERROR 或 IF @@ROWCOUNT, 用来决定是否取消从特定的存储点后接下来完成的工作。另外, 应用程序 (或存储过程) 很可能更为复杂并执行有用的工作。但是, 示例确实演示了存储点如何用于取消由事务处理中的语句所完成的部分工作, 如图 5.3 所表明的那样。

请注意, ROLLBACK 删除的只是在以下语句之后执行的两条 DELETE 语句、INSERT 语句和 UPDATE 语句的效果:

```
SAVE TRANSACTION savepoint_2
```

要了解的重要事情是, SAVE TRANSACTION 语句允许在组成事务处理的语句序列中标记存储点。然后就可使用 ROLLBACK 语句和指定的存储点取消在特定的存储点以后执行的语句, 恢复数据库。换句话说, 可以 ROLLBACK 到事务处理内的任何存储点, 然后从该处继续执行 SQL 语句。

注意: 虽然 SAVE TRANSACTION 语句允许在事务处理内命名存储点, 但并不提交存储点前所完成的工作。由此, DBMS 将整个 BEGIN TRANSACTION 语句之后的语句序列看作是单个的事务处理。因而, 如果程序发送的语句中止运行, 或者出现了其他类型的硬件或软件故障, DBMS 将取消事务处理内所有语句的效果——包括那些在存储点前和后的语句。

技巧 97 在 MS-SQL Server 上使用命名的和嵌套的事务处理

在技巧 96 “理解 MS-SQL Server 的事务处理模型”中, 读者已经学习了如何使用 SAVE TRANSACTION 语句创建允许取消事务处理内所完成的部分工作的存储点。MS-SQL Server 的提交命名的事务处理的能力可用作完全不同的目的。命名的事务处理和命名的提交应该只用于支持存储过程中的事务处理过程, 这种存储过程既可作为部分事务处理被调用, 也可在事务处理外被调用。存储过程是在事务处理内被调用还是在事务处理外被调用对于在存储过程内命名的 COMMIT 执行的语句的效果是重要的。

例如，假设有一个由以下语句创建的存储过程：

```
CREATE PROCEDURE SP_Insert_Row
    @row_number SMALLINT, @description CHAR(35)
AS BEGIN TRANSACTION insert_trans
    INSERT INTO trans_table
        VALUES (@row_number, @description)
    COMMIT TRANSACTION insert_trans
```

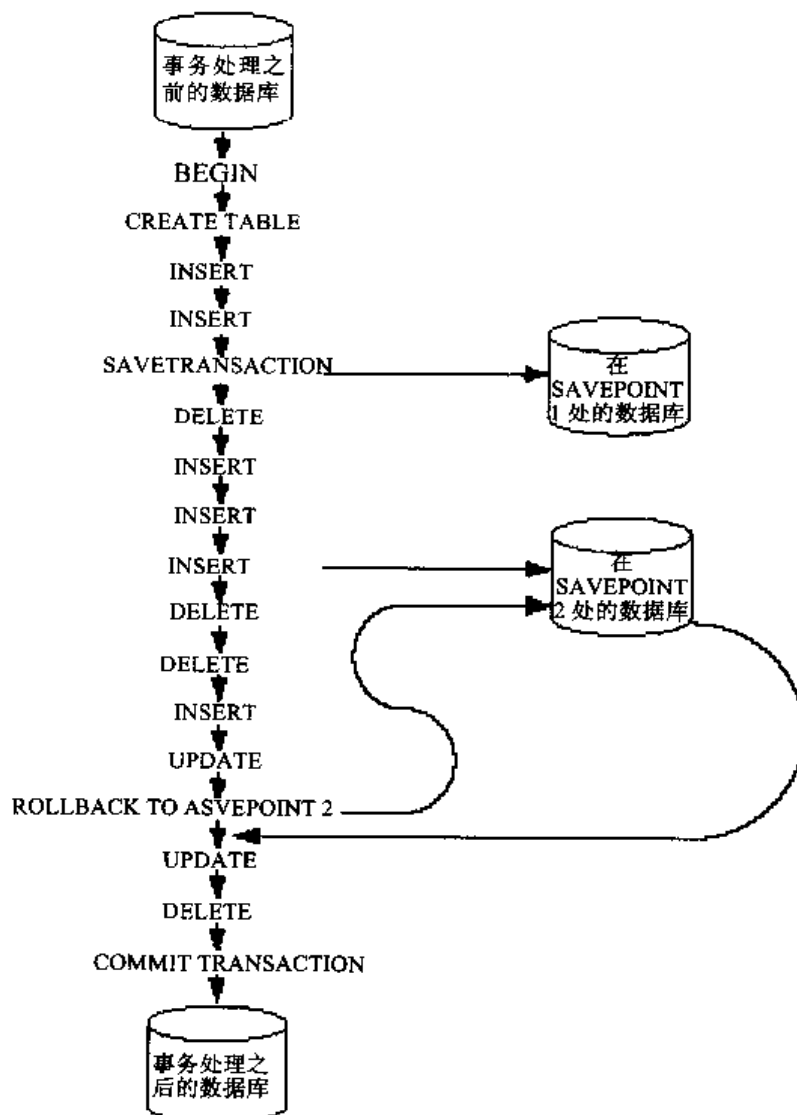


图 5.3 使用存储点的 MS-SQL Server 事务处理过程

此存储过程开始一个事务处理，插入传递到 TRANS_TABLE 中的值，然后执行一条命名的 COMMIT 语句通过终止事务处理使插入成为永久的。

如果执行以下 SQL 语句：

```
CREATE TABLE trans_table
    (row_number SMALLINT, description VARCHAR(35))
```

```
BEGIN TRANSACTION
```

```

INSERT INTO trans_table VALUES (1, 'Inserted Row 1')
INSERT INTO trans_table VALUES (2, 'Inserted Row 2')
EXEC SP_Insert_Row 3, 'Inserted Row 3'
EXEC SP_Insert_Row 4, 'Inserted Row 4'
ROLLBACK

```

TRANS_TABLE 将不会出现错误！

读者可能会期望由存储过程插入的行在 ROLLBACK 之后仍然存在，因为存储过程在每次插入之后执行以下的命名的 COMMIT 语句：

```
COMMIT TRANSACTION insert_trans
```

但是，在本例中，存储过程是在另一事务处理之内调用的，这就使得存储过程中的事务处理成为嵌套的或内部事务处理。MS-SQL Server 忽略在嵌套的（内部）事务处理内执行的命名的 COMMIT 语句。

这样一来，当在另一事务处理内调用时，最后的由例程内嵌套的（内部）事务处理所完成的工作的处理要由对外部事务处理所发生的情况来决定。在本例中，ROLLBACK 语句终止了外部的事务处理。结果，在外部事务处理内所完成的所有工作，包括由执行的例程所完成的工作都被 ROLLBACK 语句删除。

另一方面，如果在事务处理外部调用同样的例程，例如：

```

EXEC SP_Insert_Row 3, 'Inserted Row 3'

BEGIN TRANSACTION
    INSERT INTO trans_table VALUES (1, 'Inserted Row 1')
    INSERT INTO trans_table VALUES (2, 'Inserted Row 2')
    EXEC SP_Insert_Row 4, 'Inserted Row 4'
ROLLBACK

```

TRANS_TABLE 将包括：

```

row_number description
-----
3          Inserted Row 3

```

在这个例子中，由第一次调用以下例程所完成的工作被事务处理尾部的 COMMIT 语句永久地写入 TRANS_TABLE：

```
EXEC SP_Insert_Row 3, 'Inserted Row 3'
```

当例程在事务处理外部执行时，例程内的事务处理不是嵌套的，而且其 COMMIT 语句使对 TRANS_TABLE 的改变成为永久的。

作为对照，在示例中由存储过程的第二次执行所完成的工作再一次被 ROLLBACK 所取消。当例程在事务处理内执行时，存储过程内的事务处理是嵌套的，而且 DBMS 在看到 COMMIT 语句时并不使 TRANS_TABLE 的变化成为永久的（因为一条在内部事务处理中执行的 COMMIT 语句被 MS-SQL Server 忽略）。

现在要了解的重要事情是，MS-SQL Server 忽略在内部或嵌套的例程中执行的命名的 COMMIT 语句。不管嵌套的事务处理出现在存储过程内还是应用程序本身之内，这都是正确的。

注意：在存储过程内使用命名的 BEGIN TRANSACTION 和命名的 COMMIT TRANSACTION 语句的重要性起初可能并不明显。如果在存储过程内不对事务处理命名，而

且使用未命名的 COMMIT 语句将其结束，在例程末尾处的 COMMIT TRANSACTION 将不仅提交（使之成为永久的）例程内的工作，而且还结束外部的事务处理。因而，在本技巧的两个例子中的存储过程 SP_INSERT_ROW 尾部出现的未命名的 COMMIT TRANSACTION 语句可使内、外事务处理都终止并防止例子尾部处的 ROLLBACK 取消 ROLLBACK 之前的多个插入成为永久所执行的工作。

第 6 章 使用数据控制语言 (DCL) 建立数据库安全性

技巧 98 理解 MS-SQL Server 标准和 Windows NT 的综合安全性

SQL DBMS 有两种管理用户 ID 的方法。某些 DBMS 产品使用在主机操作系统中定义的用户名作为数据库认证的 ID。而别的产品，特别是那些有适用于几种不同操作系统版本的产品，维护其自己的内部用户 ID 清单和密码。MS-SQL Server 允许使用这两种安全性方法之一来管理用户账户。

所选的安全性方法控制着 MS-SQL Server 在服务器上如何管理用户账户和 DBMS 如何与 Windows NT 的安全性系统相互作用。在“标准”安全性模式下，MS-SQL Server 将接受的用户名作为认证 ID 而且还允许在 DBMS 本身之内定义附加的用户 ID/密码对。同时，在“集成的”安全性模式下，MS-SQL Server 通过操作系统主访问控制列表 (ACL) 依赖 Windows NT 来管理用户连接。

注意：MS-SQL Server 的综合安全性（允许操作系统管理用户账户）仅当在 Windows NT 平台上运行 MS-SQL Server 时才是可用的。如果正在使用其他操作系统，MS-SQL Server 必须使用标准安全性并管理其自己的用户 ID 和密码。

配置 MS-SQL Server 的最常用的方法是以标准模式启动 MS-SQL Server，让 DBMS 负责管理和维护数据库用户账户。在标准模安全性式下，MS-SQL 服务器通过对照存储在内部系统表中的信息检查用户 ID/密码对认证用户登录。结果，如果 MS-SQL Server 正在网络服务器上运行在标准完全性模式下，启动 DBMS 会话需要两步登录——一步登录到服务器上，以获得网络访问权，而第二步登录到数据库上。

另一方面，集成的安全性模式需要单步登录，既获得网络也获得数据库访问权。MS-SQL Server 允许 NT 操作系统认证用户 ID 和密码。然后创建受信任的与 DBMS 的连接——使用键入的登录名获得对网络的访问权并作为用户 ID 启动数据库会话。

为了使用 MS-SQL Server Enterprise Manager 来设置 MS-SQL Server 的安全性模式，请执行以下步骤：

1. 为了启动 Enterprise Manager，可单击 Start 按钮，将鼠标指针移动到 Start 菜单的 Programs 上并选择 Microsoft SQL Server 7.0，然后单击 Enterprise Manager。
2. 为了显示 SQL 服务器的列表，可单击 SQL Server Group 左边的加号 (+)。
3. 单击想要管理的 SQL 服务器安全系统的图标。例如，为了处理名为 NVBIZNET2 的 SQL 服务器上的安全系统，可单击 NVBIZNET2 的图标。
4. 选择 Action 菜单上的 Properties 选项。
5. 单击 Security (安全性) 选项卡。Enterprise Manager 将显示 SQL Server Properties 对话框上的安全性选项卡，如图 6.1 所示。
6. 为了使用“标准”安全性，既允许 MS-SQL Server 维护的用户 ID 也允许 Windows NT 管理的用户名来访问 SQL 服务器，可单击 SQL Server and Windows NT 左边的单选钮。为了使

用“集成的”安全性，只允许由 Windows NT 定义的用户名访问 SQL 服务器，可单击 Windows NT Only（只有 Windows NT）单选按钮。对于当前的示例来说，通过单击 SQL Server and Windows NT 单选按钮让 MSSQL Server 使用标准安全性。

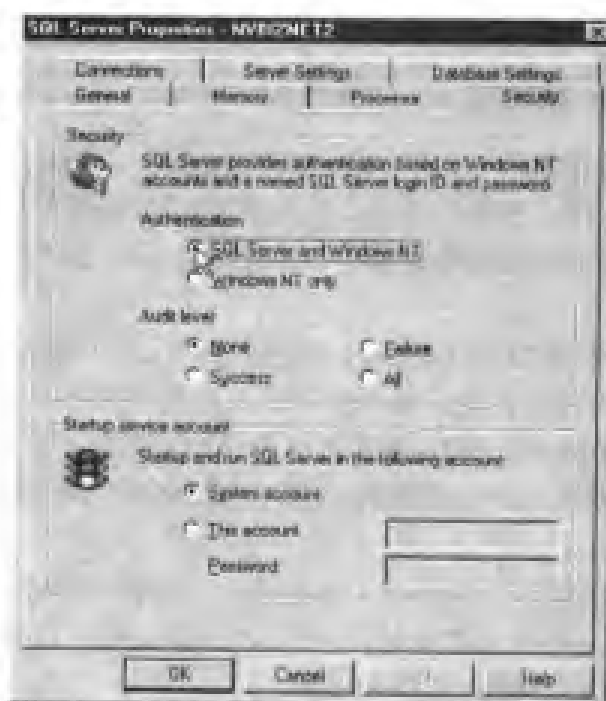


图 6.1 MS-SQL Server SQL Server Properties 对话框的 Security 选项卡

7. 单击 OK 按钮，应用程序安全性模式选择并关闭 SQL Server Properties 对话框。

不管是使用标准安全性让 MS-SQL Server 处理用户 ID 登录和密码认证还是选择综合安全性模式允许 Windows NT 操作系统来处理，都必须在 MS-SQL Server 上为允许开始数据库会话的每个认证 ID（用户名）建立用户 ID。技巧 99“使用 MS-SQL Server Enterprise Manager 添加登录和用户”中，读者将学习如何使用 MS-SQL Server Enterprise Manager 来创建和删除数据库登录和用户账户。

技巧 99 使用 MS-SQL Server Enterprise Manager 添加登录和用户

MS-SQL Server 上的用户 ID 由两个组分组成：一个登录（login），允许用户 ID 与 MS-SQL Server 连接起来；另一个是用户（user），MS-SQL Server 使用它来控制用户 ID 在给定的由 SQL Server 所管理的数据库上的权限。MS-SQL Server 将用户 ID 的登录组分存储在 MASTER 数据库的 SYSLOGINS 表中，而将用户组分存储在用户 ID 具有访问权限的每个数据库的 SYSUSERS 表中。

允许与 MS-SQL Server 连接的每个用户 ID 在 SQL 服务器的 SYSLOGINS 表中将有单一的登录条目。由于单个 MS-SQL Server 通常管理几个数据库，单一用户 ID 可能在一个或许多个数据库中具有用户组分。将用户 ID 分成登录和用户组分允许单个用户 ID 在不同的数据库上有不同的权限，而仍然只有一个密码。

这样一来，MS-SQL Server 通过认证用户 ID 的登录组分及其密码实现了 SQL 的安全性架构并建立一个会话。接着服务器使用用户 ID 的用户组分来确定对由服务器管理的数据库中的

对象是允许还是禁止执行 SQL 语句。

为了使用 MS-SQL Server Enterprise Manager 来添加用户 ID 的登录和用户组分, 请执行以下步骤:

1. 为了启动 Enterprise Manager, 可单击 Start 按钮, 将鼠标指针移动到 Start 菜单的 Programs 上并选择 Microsoft SQL Server 7.0, 然后单击 Enterprise Manager。
2. 为了显示 SQL 服务器的列表, 可单击 SQL Server Group 左边的加号 (+)。
3. 单击想要创建用户 ID 的 SQL Server 左边的加号 (+)。例如, 如果想要管理名为 NVBizNet2 的 SQL Server 上的用户 ID, 可单击 NVBizNet2 图标左边的加号 (+)。Enterprise Manager 将显示包括在所选择的 MS-SQL Server 上可用的数据库和服务的文件夹。
4. 单击 Security 文件夹左边的加号(+), Enterprise Manager 将显示在步骤 3 中所选的 SQL Server 上的安全性对象的清单, 如图 6.2 所示。



图 6.2 MS-SQL Server 的 Enterprise Manager 安全对象和登录名清单

5. 单击 Logins 图标。
 6. 选择 Action 菜单上的 New Login (新登录) 选项, 或单击 Enterprise Manager 标准工具栏上的 New Login 按钮 (从右数第二个图标), Enterprise Manager 将显示 SQL Server Login Properties-New Login (登录属性—新登录) 对话框, 如图 6.3 所示。
 7. 在 Name 域中键入新的用户 ID。对本例来说, 键入 SQLTips。
 8. 如果想要使用集成的 Windows NT 认证方法用步骤 7 中键入的用户 ID 登录到 MS-SQL Server, 可单击 Windows NT Authentication 左边的单选按钮。然后在 Domain 域中键入用户名定义其上的服务器的域名。对本例来说, 使用标准的 SQL Server 认证方法, 因而单击 SQL Server Authentication 左边的单选按钮并在 Password 域中键入密码。对本例来说, 键入 1001 作为密码。
- 注意: 如果打算对一个用户 ID 使用 Windows NT 认证方法, 一定要选择 Windows NT Authentication 并在 Domain 域中键入正确的域名。以后在没有删除该用户 ID 并重新键入的情况下不能改变一个用户 ID 的用户名或认证方法。如果选择了 SQL Server Authentication, 在 MS-SQL Server 给予该用户 ID 对数据库表的访问权之前, 用户必须键入用户 ID 和密码——即使 Windows NT 有相同的用户 ID/密码对也是如此。
9. 为了选择建立会话时用户 ID 将使用的默认数据库, 可单击 Database (数据库) 右边的下拉按钮并从由该 SQL Server 管理的数据库清单中选择默认数据库。对于本例来说, 选择

SQLTips。

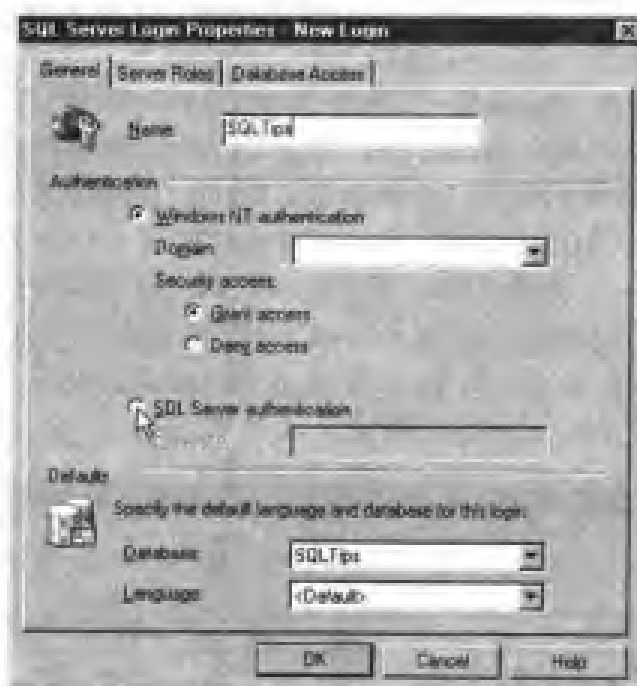


图 6.3 MS-SQL Server 的 SQL Server Login Properties-New Login 对话框的 General 选项卡

10. 单击 Database Access(数据库访问)选项卡,Enterprise Manager 将显示由该 SQL Server 管理的数据库清单,如图 6.4 所示。



图 6.4 MS-SQL Server 的 SQL Server Login Properties-New Login 对话框的 Database Access 选项卡

11. 为了给予该用户 ID 对在 SQL Server 上的数据库的访问权,可单击该用户 ID 要访问的数据库左边的复选框。对于本例来说,单击 NORTHWIND、SQLTips 和 PUBS 复选框使之出现选择标记。

12. 单击 OK 按钮。如果在步骤 8 中选择了 SQL Server Authentication, Enterprise Manager, 将显示 Confirm Password (确认密码) 对话框。重新键入在步骤 8 中键入的密码。对于本例来说, 键入 1001 并单击 OK 按钮。

13. 如果有附加的用户 ID 要定义, 从步骤 6 继续。否则单击 Enterprise Manager 窗口右上角的关闭按钮 (×), 退出 Enterprise Manager。

添加新用户 ID 的步骤 6~步骤 9 通过指定用户或应用程序用来与 SQL Server 连接的登录名建立起用户 ID 的登录组分和服务端用来认证登录的密码 (如果在步骤 8 中选择了 Windows NT Authentication 方法, MS-SQL Server 将假定操作系统认证的登录密码而且在建立与 SQL Server 连接时不再询问用户键入登录名或密码)。

虽然步骤 6~步骤 9 建立了用户 ID 的登录组分, 但步骤 10 和步骤 11 告诉服务器在 NORTHWIND、SQLTips 和 PUBS 数据库的 SYSUSERS 表中添加 SQLTips 用户 ID 的用户组分。

技巧 100 使用 MS-SQL Server Enterprise Manager 删除登录和用户

正如读者在技巧 99 “使用 MS-SQL Server Enterprise Manager 添加登录和用户” 中所学习的, MS-SQL Server 在其系统表中为添加到系统的每个用户 ID 创建登录和用户条目。由此, 通过从数据库的 SYSUSERS 表中删除一个用户 ID 的用户记录, 就可以防止该用户 ID 处理特定数据库中的对象。或者, 如果想要防止一个用户 ID 访问所有数据库中的对象并停止该 ID 与数据库服务器的连接, 可将用户 ID 的登录记录从 MASTER 数据库的 SYSLOGINS 中删除。

MS-SQL Server 允许通过执行存储过程 sp_droplogin 和 sp_dropuser 删除登录和用户, 也允许使用 Enterprise Manager 执行以下步骤达到同样的目的:

1. 按技巧 99 中的步骤 1~步骤 5 启动 Enterprise Manager 并显示系统登录名。
2. 如果想要删除登录记录, 可跳过步骤 7, 否则为了删除用户记录, 可找出想要在 Enterprise Server 窗口的右窗格中修改用户组分的用户 ID。然后右击名称 (用户 ID)。对于本例来说, 右击在技巧 99 中创建的用户 ID, 即 SQLTips。
3. 当 Enterprise Server 显示右键弹出菜单时, 选择 Properties 选项。Enterprise Server 将显示 SQL Server Login Properties (SQL Server 登录属性) 对话框, 如图 6.3 所示。
4. 单击 Database Access (数据库访问) 选项卡显示由 MS-SQL Server 管理的数据库清单 (读者已经在图 6.4 中看到过这个选项卡)。
5. 为了防止一个用户 ID 访问数据库, 可清除该数据库左边的复选框图标。对于本例来说, 单击 NORTHWIND 数据库图标左边的复选框, 使其选择标记消失。

注意: 如果一个用户 ID 在某一数据库中拥有对象 (如表、视图或存储过程), 则不能从任何数据库的 SYSUSERS 表中删除该用户 ID 的用户组分。为了防止该用户 ID 访问数据库, 可将该用户 ID 对象的拥有权转移到另一用户上, 然后清除 Permit 列上的复选框。

6. 单击 OK 按钮让 Enterprise Server 从 SYSUSERS 表 (在本例中是 NORTHWIND 数据库的 SYSUSERS 表) 中删除用户记录。

7. 为了删除登录, 在 Enterprise Server 窗口的右窗格中找出想要删除登录组分的用户 ID。然后单击名称 (用户 ID) 选择。接着, 按 Delete 键 (或者右击该名称并从弹出菜单中选择 Delete 选项)。Enterprise Server 将显示 Are You Sure You Want to Remove This Login? (确实要删除这个登录吗?) 消息框。

注意：不能删除在数据库服务器上拥有对象的任何用户 ID 的登录记录。因而，如果有拥有数据库对象（如视图、表或存储过程）的用户 ID，可用以下方法取消其登录访问权：

- 和另一用户 ID 转移对象拥有权，然后重复删除登录的步骤。
- 右击登录名，在弹出菜单中选择 Properties 选项，然后在 SQL Server Login Properties 对话框上的 General 选项卡中改变用户 ID 的密码。
- 右击登录名，在弹出菜单中选择 Properties 选项，然后在 SQL Server Login Properties 对话框上的 General 选项卡上单击 Deny Access（取消访问权）右边的单选按钮。

8. 单击 Yes 按钮确认登录删除请求。或者，如果不想删除用户名，可单击 No 按钮。如果有另外的想要删除的用户记录，可从步骤 2 处继续，或者，如果想要删除另一登录记录可从步骤 7 处继续。否则，单击 Enterprise Manager 窗口右上角的关闭按钮（×），退出 Enterprise Manager。

技巧 101 理解 MS-SQL Server 的安全角色和组用户安全性

用户组让数据库管理员（DBA）通过向几个用户同时授予（GRANT）或撤消（REVOKE）对多个数据库对象的访问权而简化权限管理。在大型的组织机构内，同一部门的雇员通常具有同样的数据库访问需求。例如，人事部门的所有雇员可能有对处理工资和雇员个人信息的相同的权限。类似地，会计部门的所有雇员很可能需要共同的一套对处理公司发票和账户支付数据表的访问权限。

ANSI/ISO 的安全模型允许以以下两种方式之一用相似的访问权限建立用户组：

- 向组中所有的人指定单一用户 ID。授予一个用户 ID 以所有用户所需的对数据库对象的访问权，允许为许多用户同时建立访问权，因为所建立的一个用户将由多个用户所使用。但是，在允许人们共享相同的用户 ID 时，不能区别一个用户登录时所完成的工作与另一个用户所完成的工作（例如，因而不能了解人事部门的哪个雇员意外地删除了 EMPLOYEE 表）。
- 为组内的每个人指定不同的用户 ID。使用多个用户 ID 允许保存每个用户负责完成的工作。但是，一遍一遍地对组内的每个成员建立同样的权限是枯燥无味的，而且易于出错——特别是当有大量的用户需要对许多数据库对象有相同的权限时。

MS-SQL Server 提供了第 3 种变种——数据库角色，允许对一个或许多数据库对象建立一套访问权限。通过为一个用户 ID 指定一个角色，就可以同时授予（GRANT）对许多数据库对象的访问权限，因而减少了（本质上是枯燥的）管理服务器（有许多数据库对象和大型的用户组需要对那些对象有相同的访问权限）上的访问权所花的时间。

有两种类型的数据库角色，标准角色和应用程序角色。标准角色就是大多数人称为组的角色，因为用户向标准角色（组）指定用户 ID，然后向标准角色授予访问数据库对象的权限。

另一方面，应用程序的角色是由密码保护的一套权限。与标准角色不同，应用程序的角色没有为其指定的用户 ID。而是应用程序使用应用程序角色名作为其用户 ID，而角色的密码用于认证，从而与 DBMS 连接。一旦连接建立起来，DBMS 就授予应用程序以应用程序角色中定义的权限。

为了使用 Enterprise Manager 向 MS-SQL Server 数据库角色指定用户 ID，请执行以下步骤：

1. 为了启动 Enterprise Manager，可单击 Start 按钮，将鼠标指针移动到 Start 菜单的

Programs 上并选择 Microsoft SQL Server 7.0, 然后单击 Enterprise Manager。

2. 为了显示 SQL 服务器的列表, 可单击 SQL Server Group 左边的加号 (+)。

3. 单击想要在其上指定数据库角色的 SQL Server 左边的加号 (+)。例如, 如果想要在名为 NVBizNet2 的 SQL Server 上指定角色, 可单击 NVBizNet2 图标左边的加号 (+)。Enterprise Manager 将显示包括在所选择的 MS-SQL Server 中可用的数据库和服务的文件夹。

4. 单击 Databases 文件夹左边的加号 (+)。Enterprise Manager 将显示在步骤 3 中所选的服务管理的数据库清单。

5. 选择想要指定角色的数据库。对本例来说, 在展开的 Databases 清单中单击 SQLTips 图标左边的加号 (+), Enterprise Manager 将显示所选数据库上可用的数据库对象的清单。

6. 为了让 Enterprise Manager 在右窗格中显示数据库角色的清单, 可在步骤 5 中展开的数据库对象清单中单击 Roles (角色) 图标。

7. 右击想要添加成员的角色, 并从弹出菜单中选择 Properties 选项。对本例来说, 在 Enterprise Manager 的右窗格中右击 DB_OWNER 角色, 然后在弹出菜单中选择 Properties 选项。Enterprise Manager 将显示 Database Role Properties (数据库角色属性) 对话框, 如图 6.5 所示。

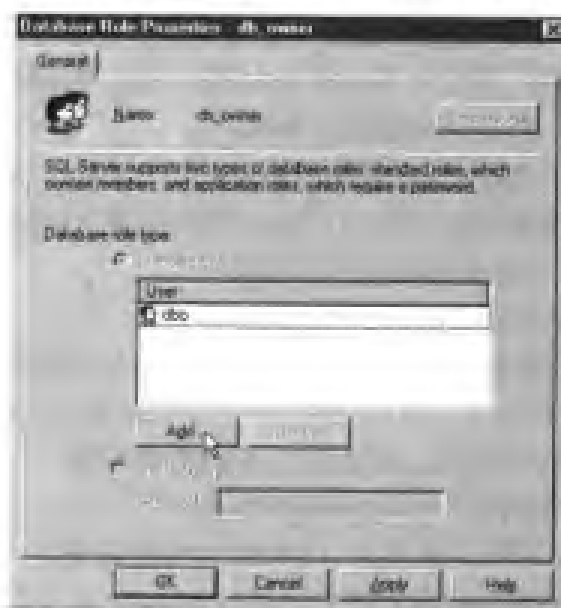


图 6.5 MS-SQL Server Enterprise Manager 的 Database Role Properties 对话框

8. 为了向该角色添加用户 ID, 可单击 Add 按钮。Enterprise Manager 将显示还没有指定给角色的用户 ID 清单。

9. 选择想要向角色添加的用户 ID, 然后单击 OK 按钮。对本例来说, 单击 SQLTips, 然后单击 OK 按钮。Enterprise Manager 将把所选的用户 ID 添加到该角色上并返回 Database Role Properties 对话框。

10. 单击 OK 按钮保存所做改变并返回 Enterprise Manager 应用程序窗口。

11. 单击 Enterprise Manager 应用程序窗口右上角的关闭按钮 (×), 退出 Enterprise Manager。

读者将在技巧 104 “使用 MS-SQL Server Enterprise Manager 创建数据库角色”中学习如何创建 MS-SQL Server 的标准和应用程序角色。

技巧 102 理解 MS-SQL Server 的权限

权限是一个用户 ID 具有的访问诸如表、视图、域或存储过程一类的数据库对象的权利。数据库管理员（DBA）通过授予及撤消用户 ID 和角色的权限控制用户及应用程序与数据库的相互作用。

除了由 DBA 明确授予的权限，DBMS 还隐含（自动）地向对象的拥有者或创建者给予全套的对象权限。由此，数据库对象拥有者（DBOO）自动地对用户或应用程序所创建的数据库对象具有 SELECT、INSERT、UPDATE、DELETE、EXEC、DRI（外键引用）和数据定义语言（DDL）权限。另外 DBOO 还可向其他用户 ID 授予（GRANT）和撤消（REVOKE）权限。

注意：当一个用户 ID 创建数据库时（与数据库内的表、视图或其他对象不同），用户 ID 变为数据库的拥有者（DBO）。MS-SQL Server 自动地给予该用户 ID 和对该数据库中的所有对象的全部权限。另外，在安装 MS-SQL Server 时自动创建的系统管理员（SA）账户有对 SQL Server 上的所有数据库中的所有对象的全部权限。

对 MS-SQL Server 上可用的数据库对象的权限如下：

- SELECT。允许用户 ID 从表或视图中查询或是读取数据。SELECT 权限可被限制到表或视图内独立的列，或作为整体授予对整个对象的权限。
- INSERT。允许用户 ID 向表或视图添加行。
- UPDATE。允许用户 ID 改变表或视图中的数据。类似于 SELECT 权限，UPDATE 权限可限制为对表或视图内独立的列，或作为整体授予对整个对象的权限。
- DELETE。允许用户 ID 从表或视图中删除行。
- EXEC。（执行）允许用户 ID 执行存储过程。
- DRI。（声明引用完整性）允许用户 ID 在表上添加 FOREIGN KEY 约束。
- CREATE TABLE。允许用户 ID 向数据库中添加新表。正如读者已经了解的，用户 ID 将成为他所创建表的 DBOO，因而有对表的所有权限。
- ALTER TABLE。允许用户 ID 改变表的结构。
- DROP TABLE。允许用户 ID 丢弃（DROP）或删除表。
- CREATE。允许用户 ID 创建数据库对象（如 DEFAULT、PROCEDURE、RULE 或 VIEW），并向其授予 ALTER 权限。正如以前所提到的，用户 ID 将成为所创建的数据库对象的 DBOO，具有对新数据库对象的所有权限。
- ALTER。允许授予此权限的用户 ID 修改数据库对象（如 DEFAULT、PROCEDURE、RULE 或 VIEW）的结构或者数据库对象内的语句（在存储过程的情况下）。
- DROP。允许授予此权限的用户 ID 删除数据库对象（如 DEFAULT、PROCEDURE、RULE 或 VIEW）。
- BACKUP DATABASE。允许用户 ID 将整个数据库备份到备份设备上。
- BACKUP LOG。允许用户 ID 将数据库事务处理日志备份到备份设备上。
- CREATE DATABASE。允许用户 ID 在 SQL Server 上创建新数据库。在创建数据库之后，该用户 ID 成为所创建数据库的 DBO，并因此具有对由任何用户 ID 创建的新数据库中所创建的所有对象的所有权限。

技巧 103 理解 SQL 的安全对象和权限

SQL 数据库由诸如表、视图、存储过程、约束、默认值、用户定义的数据类型、索引和用户 ID 一类的对象组成。DBMS 要施以安全保护的进一步区分为安全性对象。由于表和视图是两种主要的数据库对象，第一个 SQL 标准 (SQL-89) 将这两者指定为安全性对象是符合逻辑的。

SQL-92 扩展了安全性对象的列表，除了表和视图之外还包括域、数据类型和字符集，并为表和视图增加了外部引用保护。简短地说，SQL 标准 (SQL-89 和 SQL-92) 规定，DBMS 一定要能够单独地保护在所有标准的 SQL 数据库产品中发现的重要对象。保护数据库安全性对象意味着允许用户 ID 有对某些对象的某种访问权限，而禁止访问另一些对象。

注意：每种 DBMS 产品都通过对其开发人员认为值得保护的数据对象提供安全保护而扩展了标准的安全性对象的清单。例如，MS-SQL Server 为存储过程增加了安全保护。同时，DB2 包括了对索引、架构和包的保护（请检查一下系统手册中的 GRANT 和 REVOKE 语句，找出该 DBMS 的具体实现中的安全性对象的完整清单）。现在要了解的重要事情是，底层 SQL 安全性架构是使用 GRANT 和 REVOKE 语句一个用户 ID 一个用户 ID 地为数据库对象授予或是撤销特定的 SQL 语句的执行权限。

用户 ID 可对数据库对象执行的一套 SQL 语句（也就是用户 ID 可对数据库中的“东西”所做的“事情”）就是用户 ID 对安全对象的权限。所有的商业 DBMS 产品都支持 4 种在 SQL-89 和 SQL-92 中定义的安全性权限：

- **SELECT**。允许用户 ID 在 SELECT 语句中的 FROM 子句中使用表或视图名从表或视图中提取数据。虽然 SQL 允许对整个表将 SELECT 权限授予或撤销，有几种 DBMS 产品（包括 SYBASE 和 MS-SQL Server）允许对表或视图中的特定列授予 SELECT 权限（与整个对象不同）。
- **INSERT**。允许用户 ID 在 INSERT 语句中的 INTO 子句中使用表名向表或视图添加新行。SQL-92 通过允许 DBMS 将此权限限制为只对每行特定的列，从而扩展了原来 SQL-89 的 INSERT 安全性权限。这样一来，一个用户 ID 可以向表中添加新行，它可能不得不使表的某些列的值为 NULL。
- **DELETE**。允许用户 ID 在 DELETE 语句中的 FROM 子句中使用表或视图名从表或视图中删除行。
- **UPDATE**。允许用户 ID 使用表或视图名作为 UPDATE 语句的目标，从而改变表或视图中的值。SQL-92 通过允许 DBMS 将此权限限制为只对每行特定的列，从而扩展了原来 SQL-89 的 UPDATE 安全性权限。因此，一个用户 ID 可以改变某些列中的值，但对其他列却无能为力。

大多数 DBMS 产品还支持添加到 SQL-92 的两种新权限：

- **REFERENCES**。允许用户 ID 在外键或检查约束中引用表或视图中的列（技巧 114 将向读者展示一个用户 ID 如何使用外键或检查约束来决定列中的值，即使该用户 ID 被撤销了对列或整个表的 SELECT 权限）。
- **USAGE**。允许用户 ID 使用域（命名的合法列值集）、用户定义的数据类型、字符集、顺序序列和转换。例如，如果没有对域的 USAGE 权限，一个用户 ID 就不能创建有

使用域名作为其数据类型的列的表。

具体的 DBMS 产品很可能支持不止这 6 种由 SQL-92 标准指定的安全权限（在技巧 102 “理解 MS-SQL Server 的权限”中读者已经学习有关几种 MS-SQL Server 支持的附加权限的内容）。要了解的重要事情是，DBMS 通过允许一个用户 ID 只执行那些授予权限的语句，从而保护其对象。请检查一下有关 GRANT 和 REVOKE 语句的系统文档，从而找出 DBMS 所支持的安全对象和权限的清单。

技巧 104 使用 MS-SQL Server Enterprise Manager 创建数据库角色

正如读者在技巧 101 “理解 MS-SQL Server 的安全角色和组用户安全性”中所学过的，可以通过将 ID 指定给“角色”从而授予一个用户 ID 多种对一个或多个安全对象的权限。事实上，角色为 MS-SQL Server 提供了组用户安全性架构。在向角色中添加安全性权限时，该权限即同时授予了所有的组成员。作为对照，从角色中删除权限会引起组中的每个用户 ID 失去权限。这样一来，将角色指定给一个用户 ID 与把 ID 放入带有角色名的组是相同的。

为了使用 MS-SQL Server Enterprise Manager 向数据库中添加角色，可执行以下步骤：

1. 为了启动 Enterprise Manager，可单击 Start 按钮，将鼠标指针移动到 Start 菜单的 Programs 上并选择 Microsoft SQL Server 7.0，然后单击 Enterprise Manager。
2. 为了显示 SQL 服务器的列表，可单击 SQL Server Group 左边的加号（+）。
3. 单击想要在其上创建数据库角色的 SQL Server 左边的加号（+）。例如，如果想要在名为 NVBizNet2 的 SQL Server 上创建角色，可单击 NVBizNet2 图标左边的加号（+）。Enterprise Manager 将显示包括在所选择的 MS-SQL Server 上可用的数据库和服务的文件夹。
4. 单击 Databases 文件夹左边的加号（+）。Enterprise Manager 将显示在步骤 3 中所选的服务器管理的数据库清单。
5. 选择想要创建角色的数据库。对本例来说，在展开的 Databases 清单中单击 SQLTips 图标左边的加号（+），Enterprise Manager 将显示所选数据库上可用的数据库对象的清单。
6. 选择 Action 菜单中的 New 选项，再单击 Database role。Enterprise Manager 将显示 Database Role Properties-New Role（数据库角色属性—新角色）对话框，如图 6.6 所示。



图 6.6 MS-SQL Server Enterprise Manager 的 Database Role Properties-New Role 对话框

7. 在 Name 域中键入新角色名。对本例来说, 键入 SQLTips_Users。

8. 选择想要创建的角色类型, 单击 Standard Role (标准角色) 左边的单选按钮, 从而创建用户 ID 组, 或者单击 Application Role (应用程序角色) 左边的单选按钮, 从而为应用程序创建权限集。对本例来说, 单击 Application Role (应用程序角色) 左边的单选按钮。

9. 如果正在创建应用程序角色, 在步骤 10 处继续, 否则为了向一个或多个用户 ID 指定标准角色, 可单击 Add 按钮。Enterprise Manager 将显示 Add Role Members (添加角色成员) 对话框, 其中显示还未指定角色的用户 ID 清单。单击想要包括在角色成员内的用户 ID, 然后单击 OK 按钮。对本例来说, 单击用户 ID SQLTips (在技巧 99 “使用 MS-SQL Server Enterprise Manager 添加登录和用户” 中创建的), 然后单击 OK 按钮。

10. 如果正在创建标准角色, 可在步骤 11 处继续。否则, 为了完成应用程序角色的创建, 在 Password 域中键入应用程序角色的密码。

注意: 除非想要任何知道应用程序角色名的用户或应用程序有对数据库的访问权, 一定要在 Password 域中键入非空密码。

11. 为了向数据库中添加新角色, 可单击 OK 按钮。

注意: MS-SQL Server 角色对所创建的数据库来说是本地的。因而, 如果在一个数据库 (例如 SQLTips 数据库) 中创建了一个角色, 当在 PUBS 数据库中指定用户 ID 时, 该角色是不可用的。由此, 必须在每一个需要使用角色的数据库中创建角色——除非是在 MODEL 数据库中创建角色。角色与 MODEL 数据库中的其他对象一样, 都要添加到 MS-SQL Server 所创建的新数据库中。

在创建了标准或应用程序角色之后, 必须授予 (GRANT) 该角色以权限, 这将在技巧 105 “使用 MS-SQL Server Enterprise Manager 指定数据库角色权限” 中学习。请记住, 一个角色可使用只指定给标准角色或应用程序组成员的用户 ID 的权限, 或使用应用程序角色名作为用户 ID 登录的用户的权限。

技巧 105 使用 MS-SQL Server Enterprise Manager 指定数据库角色权限

读者在技巧 104 “使用 MS-SQL Server Enterprise Manager 创建数据库角色” 中已经学习了如何创建标准和应用程序角色, 而在技巧 101 “理解 MS-SQL Server 的安全角色和组用户安全性” 中学习了如何向用户 ID 指定标准角色。但是, 正如在技巧 104 中所提到的, 一个角色将使用户 ID 只被指定那些组成角色权限集的权限。有两种方法可给予角色以它所能给予其成员的权限集——既可使用 GRANT 语句, 也可使用 Enterprise Manager 编辑角色属性。

为了给予一个角色 (而且通过扩展, 即指定其上的用户 ID) 以访问数据库安全对象的权限, 可执行以下步骤:

1. 执行技巧 101 中的步骤 1~步骤 6, 启动 Enterprise Manager 并显示 Database Role Properties (数据库角色属性) 对话框。

2. 右击想要指定权限的角色。对本例来说, 右击在技巧 104 中创建的 SQLTips_USERS Role 图标。Enterprise Manager 将显示 Database Role Properties 对话框。

3. 为了显示如图 6.7 中所示的 Permissions 选项卡, 可单击 Permissions (权限) 按钮。



图 6.7 MS-SQL Server Enterprise Manager 的 Database Role Properties-New Role 对话框的 Permissions 选项卡

注意：如果只改变过去曾经给予角色以权限的对象的某些权限，可单击 List Only Objects with Permissions for This Role（只列出有此角色权限的对象）左边的单选按钮，以减少显示在 Permissions 选项卡的对象清单区的数据库对象数目。当想要给予角色以另外的数据库对象的权限时（本例中的情况就是这样），最好在清单区看到所有的数据库对象，这是默认情况。

4. 使用数据库对象清单右边的滚动条找出想要的角色具有 SQL 语句执行权的数据库对象。对本例来说，找出在前面技巧中创建的任意一个表。如果没有创建任何表，可使用视图之一（在 Object 列上由眼镜图标标示的）。

5. 为授予（GRANT）权限，可单击想让用户 ID 或应用程序指定角色的权限列上的复选框。例如，对 STUDENTS 表授予（GRANT）SELECT、INSERT、UPDATE、DELETE 和 DRI 权限，可单击列表框的 STUDENTS 行的列的复选框，使之出现选择标记。

注意：并不是所有权限对所有数据库对象都是可用的，如果权限对数据库对象不可用，则找不到该列的权限复选框。例如，图 6.7 显示出只能对 SP_INSERT_ROW 可授予（GRANT）EXEC（执行）权限。这是有道理的，因为 SP_INSERT_ROW 是个存储过程。其他权限对它不可用。作为对照，不能授予角色以对在对象区中列出的表或行的 EXEC 权限。最后 DRI（外部 FOREIGN KEY 和 CHECK 引用权限）只对数据库表才是可用的。

6. 重复步骤 4 和步骤 5，直到已经选择所有想要让角色（及其成员和应用程序）具有的对每个数据库对象的权限。

7. 单击 OK 按钮保存权限选择并返回 Database Role Properties 对话框的 General 选项卡。

8. 为了更新在数据库系统表中角色的定义，可单击 Database Role Properties 对话框上的 OK 按钮。

9. 为了退出 Enterprise Manager，单击 Enterprise Manager 窗口右上角的关闭按钮（×）。

在使用 Enterprise Manager 向角色授予（GRANT）语句执行权限中有一个小问题，只允许

对整个数据库对象授予 (GRANT) 或撤消 (REVOKE) 每项权限。由此, 如果使用 Enterprise Manager 向 SQLTips_USERS 角色授予 (GRANT) 对 EMPLOYEES 表的 UPDATE 权限, 所有的角色成员都能够修改 EMPLOYEES 表中的所有列。

在技巧 106~技巧 117 中, 读者将学习如何使用 GRANT 语句来限制角色对表或行中特定列的权限。

技巧 106 使用 GRANT 语句的 WITH GRANT OPTION 允许用户向其他用户授予对数据库对象的访问权

作为数据库对象的拥有者 (DBOO), 可以控制哪些用户 ID 可访问所创建的对象以及用户可对那些对象做些什么。创建了诸如表或视图之类的对象之后, 只有在使用 GRANT 语句向另外的用户授予访问权限以后, 别人才能对此对象执行 SQL 语句。

注意: 数据库管理员 (DBA) 和数据库拥有者 (DBO) 账户有对数据库中的所有对象的全部权限, 包括 GRANT 权限。

例如, 如果在创建了 EMPLOYEES 之后立即执行以下 GRANT 语句:

```
GRANT ALL PRIVILEGES ON employees TO sally
```

此时只有自己的用户 ID 和用户 ID SALLY 能够处理 EMPLOYEES 表。另外, 虽然 GRANT 语句给予 Sally 以对 EMPLOYEES 表的所有权限, 她也不能给予其他用户 ID 以访问表的权限, 因为上面的 GRANT 语句未包括 WITH GRANT OPTION 子句。

例如, 假设刚创建了名为 VW_VEGAS_EMPLOYEES 的 Las Vegas 的雇员视图。为了给予 Las Vegas 办公室经理 Mary (用户 ID 为 MARY) 以查询访问该视图的权限, 可执行以下 GRANT 语句:

```
GRANT SELECT ON vw_vegas_employees TO mary
```

在授予她以 SELECT 权限之后, Mary 就能够查询为 VW_VEGAS_EMPLOYEES 视图提供数据的底层表中的数据。但是, 她不能再向别人授予处理视图的权限。另外, Mary 不能创建基于 VW_VEGAS_EMPLOYEES 视图的另外视图。

注意: DBOO 具有对其所创建的对象的所有权限 (包括 GRANT)。但是, 在本例中, 当 Mary 不具有对该视图的 WITH GRANT OPTION 访问权时, 如允许 Mary 创建一个对象也就允许另外的用户看到 VW_VEGAS_EMPLOYEES 视图数据, 这就侵犯了 DBMS 的访问安全性。因此, DBMS 防止一个用户 ID 创建基于对其没有 SELECT 权限以及 WITH GRANT OPTION 的任何对象 (表或视图) 的视图。

如果想让 Mary 能够向其他用户 ID 授予 (GRANT) 对 VW_VEGAS_EMPLOYEES 的 SELECT 权限, 可包括 WITH GRANT OPTION。例如, 以下 GRANT 语句中的 WITH GRANT OPTION:

```
GRANT SELECT ON vw_vegas_employees  
TO mary WITH GRANT OPTION
```

允许要授予权限列表的用户 ID 将列表中的任何权限传递给另外的用户。这样一来, 在本例中的 WITH GRANT OPTION 允许 Mary 向另外的用户 ID 授予 (GRANT) 对 VW_VEGAS_EMPLOYEES 的 SELECT 权限。但是 Mary 不能授予 (GRANT) 别人以她还没有通过 WITH GRANT OPTION 获得的任何权限 (在本例中, 如 INSERT、DELETE、UPDATE

或 REFERENCES)。

当在 GRANT 语句中包括 WITH GRANT OPTION 时一定要小心。通过允许另外的用户给予对数据库对象的访问权，信任他或她应该像保护数据库对象不受其他用户侵扰一样小心。如果给予另外的用户以 WITH GRANT OPTION 权限，则该用户不仅可授予 (GRANT) 他或她所具有的对对象的权限，而且还可将权限和 WITH GRANT OPTION 权限传递给另外的用户 ID。

技巧 107 理解 REVOKE 语句

数据库对象所有者 (DBOO) 可使用 GRANT 语句给予其他用户以对所拥有的对象的访问权限。REVOKE 语句正好做相反的事。执行一条 REVOKE 语句，DBOO 就将其先前所给予其他用户 ID 和 (或) 角色的对数据库对象的访问权限取消。

REVOKE 语句的定义为：

```
REVOKE [GRANT OPTION FOR]<privilege list>[(<column list>)]  
ON <object name>  
FROM <name list> [CASCADE | RESTRICT]
```

如果先前通过以下的 GRANT 语句向用户 ID FRANK 授予了对 EMPLOYEES 表的访问权限：

```
GRANT SELECT, INSERT, REFERENCES ON employees TO frank
```

然后执行以下 REVOKE 语句：

```
REVOKE INSERT, REFERENCES ON employees FROM frank
```

用户 ID FRANK 将只保留对 EMPLOYEES 表的 SELECT 权限。

当执行 REVOKE 语句时，请记住，只可取消以前授予的权限。因此，如果用户和另一用户向一个用户 ID 都授予了对对象的相同权限，如果用户后来撤消 (REVOKE) 了权限而另一用户却没有撤消，则该用户 ID 仍然拥有权限。

例如，假设用户 ID CAROL 执行以下 GRANT 语句：

```
GRANT SELECT, INSERT, UPDATE ON employees TO frank
```

而读者执行了以下 GRANT 语句：

```
GRANT SELECT, INSERT, REFERENCES ON employees TO frank
```

如果读者后来执行了以下 REVOKE 语句：

```
REVOKE ALL PRIVILEGES ON employees FROM frank
```

则用户 ID FRANK 将仍然拥有对 EMPLOYEES 表的 SELECT、INSERT 和 UPDATE 权限，因为那些权限是由 CAROL 授予的。读者的 REVOKE ALL PRIVILEGES 命令只删除了自己以前授予 FRANK 的所有对 EMPLOYEES 的权限。

技巧 108 使用带 CASCADE 选项的 REVOKE 语句删除权限

正如读者在技巧 106 “使用 GRANT 语句的 WITH GRANT OPTION 允许用户给予其他用户以数据库对象访问权”中所学习的，当数据库对象所有者 (DBOO 或其他授权者) 向 GRANT 语句添加 WITH GRANT OPTION 时，接受对对象的权限的用户 ID 可向其他用户授予 (GRANT) 那些权限。向 REVOKE 语句添加 CASCADE 选项允许原来的授权者从初始接受者以及权限所传递到的其他用户 ID 身上取消授予的权限。

例如，以下 GRANT 语句：

```
GRANT SELECT, INSERT ON employees TO frank  
WITH GRANT OPTION
```

给予用户 ID FRANK 以对 EMPLOYEES 表的 SELECT 和 INSERT 权限。WITH GRANT OPTION 给予 FRANK 以使用以下 GRANT 语句将其拥有的权限(包括 WITH GRANT OPTION) 授予另外的用户 ID:

```
GRANT SELECT, INSERT ON employees to SUE WITH GRANT OPTION.
```

由于用户 ID SUE 从 FRANK 处接受了权限和 WITH GRANT OPTION, 她还可执行以下 GRANT 语句:

```
GRANT SELECT ON employees TO scott
```

如果一个用户想要撤消一个或多个以前向另一用户授予的权限, REVOKE 语句上的 CASCADE 选项告诉 DBMS 删除从原始的 GRANT 语句生成的所有权限。因而, 在本例中, 以下 REVOKE 语句:

```
REVOKE INSERT ON employees FROM frank CASCADE
```

将从用户 ID FRANK 身上取消对 EMPLOYEES 表的 SELECT 权限, 然后一层一层地通过安全系统从 SUE 和 SCOTT 身上删除对 EMPLOYEES 表的 SELECT 权限(因为 SELECT 权限从原来授予 FRANK 的向下传递)。

SQL-92 要求在取消已经向其他用户授予的权限时, 告诉 DBMS 想要做什么。MS-SQL Server 实现了该标准, 要求向删除以前通过 WITH GRANT OPTION 授予的权限的 REVOKE 语句中添加 CASCADE 选项。因而, 如果正在运行 MS-SQL Server, 就不能只删除原来的用户 ID 的权限。由于 REVOKE 语句必须包括 CASCADE 选项, MS-SQL Server 要求不仅删除以前从授予权限的用户 ID 上的权限, 而且还要删除所有授予权限的用户 ID 上的权限。

由于不同的实现处理撤消从一个用户传递到下一用户的权限的方式不同, 因而必须检查系统文档上的 REVOKE 语句, 看一看具体的 DBMS 要求如何处理用 WITH GRANT OPTION 授予的权限的删除过程。

如果授予权限的用户 ID 还没有将权限传递给另一个用户, 某些 DBMS 产品允许发出没有 CASCADE 选项的 REVOKE 语句。其他产品提供 RESTRICT 选项, 以至于以下 REVOKE 语句:

```
REVOKE INSERT ON employees FROM frank RESTRICT
```

如果 FRANK 还没有将对 EMPLOYEES 表的 INSERT 权限授予另一个用户, 则上面语句将成功执行, 而当 FRANK 已经向另一个用户授予了权限, 则上面语句将失败(出现一条错误消息)。

保护数据库对象(特别是表和视图)不受未经授权访问对 DBMS 来说是很重要的。因此, 最好保持对谁能对自己拥有的对象做什么紧密的控制。因而, 当撤消以前使用 WITH GRANT OPTION 授予的权限时, 一定要添加 CASCADE 选项。如果 DBMS 只允许撤消原来用户的权限, 则会丧失对谁对自己拥有的数据库对象具有访问权限的控制。毕竟, 实际上并不知道想要撤消访问权限的用户已经使用了 GRANT 语句的 WITH GRANT OPTION 向另外的什么人授予了权限。

注意: 几乎所有的 DBMS 产品都自动地撤消从原来的 GRANT 语句派生出来的所有权限。由此, REVOKE 语句中的 CASCADE 选项只是提醒人们, 执行了 REVOKE 语句可能不止是防止语句中的指定的用户 ID 的数据库对象访问, 而且可能还会防止其他人对数据库对象的访问。

技巧 109 使用 REVOKE 语句的 GRANT OPTION FOR 子句删除 GRANT 权限

执行包括 WITH GRANT OPTION 的以下 GRANT 语句：

```
GRANT SELECT, INSERT, UPDATE, DELETE ON invoices  
TO konrad, mary WITH GRANT OPTION
```

给予语句中指定的用户 ID 以将其接受的任何或所有权限向其他用户传递的能力。这样一来，在本例中，用户 ID KONRAD 和 MARY 可以向其他用户 ID 或角色（用户组）授予对 INVOICES 表的 SELECT、INSERT、UPDATE 和 DELETE 权限。

在技巧 108 “使用带 CASCADE 选项的 REVOKE 语句删除权限”中，读者学习了如何使用带 CASCADE 选项的 REVOKE 语句删除一个用户 ID 的权限以及通过实施 WITH GRANT OPTION 授予其他人的权限。但是，有时想要只删除该用户 ID 的 GRANT 权限，而保留对对象的实际访问权限不变，这正是 REVOKE 语句中的 GRANT OPTION FOR 子句有用武之地的地方。

REVOKE 语句中的 GRANT OPTION FOR 子句允许删除一个用户 ID 的向其他用户授予对数据库对象的权限的能力，而不会影响其对数据库对象的访问权限。

例如，在 MS-SQL Server 上执行以下 REVOKE 语句：

```
REVOKE GRANT OPTION FOR INSERT ON invoices  
FROM konrad CASCADE
```

将允许用户 ID KONRAD 保留对 INVOICES 表的 SELECT、INSERT、UPDATE 和 DELETE 权限，但撤销 KONRAD 授予其他用户的 INSERT 权限并防止 KONRAD 今后向别人授予 INSERT 权限。因而，如果 KONRAD 以前执行了以下 GRANT 语句：

```
GRANT INSERT, UPDATE ON invoices TO sue  
GRANT SELECT, INSERT, UPDATE ON invoices TO frank
```

当数据库对象的拥有者（DBOO）执行了以下 REVOKE（GRANT OPTION FOR）语句之后，SUE 将保留对 INVOICES 表的 UPDATE 权限，而 FRANK 将保留 SELECT 和 UPDATE 权限：

```
REVOKE GRANT OPTION FOR INSERT ON invoices  
FROM konrad CASCADE
```

虽然在撤销一个用户 ID 的 GRANT 权限时，MS-SQL Server 撤销了传递给其他用户的权限，但并不是所有的 DBMS 产品都有同样的行为。

与撤销权限不同，此时告诉 DBMS 从用户 ID 以及向其传递了权限的任何用户身上撤销权限，撤销 WITH GRANT OPTION 只是告诉 DBMS 不允许用户 ID 及其传递了 WITH GRANT OPTION 的用户再向别人授予权限。MS-SQL Server 通过取消在语句中指定的用户 ID 的 GRANT 权限，同时撤销由执行 GRANT 选项而传递给其他用户 ID 的权限，从而实现了 GRANT 权限的撤销。

请检查自己的系统文档上的 REVOKE 语句，看一看具体的 DBMS 是如何处理撤销用户 ID 向其他用户授予（GRANT）访问权限的能力的。所有的 DBMS 产品允许人们使用 REVOKE GRANT OPTION 语句取消一个用户 ID 的向其他人授予列在语句中的权限的能力（取消了 GRANT 选项的用户 ID 仍然保留着他自己的对对象的访问权限）。某些产品（与 MS-SQL Server 不同）还允许以前授予了权限的用户（其 GRANT 权限正要撤销）继续实施以前从原来的 GRANT WITH GRANT OPTION 派生来的权限。

技巧 110 使用 GRANT SELECT (以及 REVOKE SELECT) 语句控制对数据库对象的访问

向一个用户 ID 或角色授予对表或视图的 SELECT 权限就允许该用户 ID 或角色成员“看到”表中的数据。数据库对象拥有者 (DBOO) 具有对其所创建的对象的全部权限, 而所有其他数据库用户完全没有访问权。因而, DBOO 必须执行一条带 SELECT 作为权限清单中的权限之一的 GRANT 语句, 以允许其他用户 ID 和程序查看表中的数据。

用于给予 SELECT 权限的 GRANT 语句的句法如下:

```
GRANT SELECT [(<column list>)]  
ON <table name> | <view name>  
TO <user and/or Role name list> [WITH GRANT OPTION]
```

这样一来, 例如, 为了允许用户 ID KONRAD 对 EMPLOYEES 表执行 SELECT 或 CREATE VIEW 语句, 表的 DBOO (或其他已经授予了 SELECT 权限的用户) 必须执行如下所示的一条 GRANT 语句:

```
GRANT SELECT ON employees TO konrad
```

或者, 为了向 MARY、SUE 和 PAYROLL_USERS 角色授予对视图 VW_LV_EMPLOYEES 的 SELECT 访问权, 视图的 DBOO 可以执行以下 GRANT 语句:

```
GRANT SELECT ON vw_lv_employees TO mary, sue, payroll_users
```

作为最后的示例, 为了给予 FRANK 以对 INVOICES 表的 SELECT 访问权并允许 FRANK 向其他用户授予 SELECT 访问权, 对 INVOICE 表具有 GRANT SELECT 权限的用户 ID 可在以下 GRANT 语句中包括 WITH GRANT OPTION:

```
GRANT SELECT ON invoices TO frank WITH GRANT OPTION
```

授予另一用户 ID 或角色的 SELECT 权限 (有或没有 GRANT SELECT 选项) 只允许用户 ID 或程序查询和查看表的数据。这并没有转移改变列值、从表中删除行或向表中添加行的能力。

当向用户 ID 或角色授予对对象的 SELECT 访问权的人不再想让该用户 ID 或角色成员查看表内的数据时, 授权者可执行 REVOKE 语句删除访问权。删除访问权限的 REVOKE 句法如下:

```
REVOKE [GRANT OPTION FOR] SELECT [(<column name list>)]  
ON <table name> | <view name>  
FROM <user and/or Role name list> [CASCADE]
```

这样一来, 如果 DBOO 向 MARY、SUE 和 PAYROLL_USERS 角色成员授予了对 VW_LV_EMPLOYEES 视图的 SELECT 访问权, 但后来只想让 SUE 有查询视图的访问权, DBOO 可执行以下 REVOKE 语句:

```
REVOKE SELECT ON vw_lv_employees FROM mary, payroll_users
```

要记住的一件重要事情是, 一个用户 ID 只可撤消 (REVOKE) 所授予的访问权限。因而, 如果 DBOO 使用下面的 GRANT 语句向用户 ID FRANK 授予了对 INVOICES 表的 SELECT 访问权:

```
GRANT SELECT ON invoices TO frank WITH GRANT OPTION
```

而 FRANK 接着又执行其 GRANT 选项向 SUE 授予了对 INVOICES 表的 SELECT 访问权, DBOO 不能使用以下 REVOKE 语句撤消 SUE 的 SELECT 访问权限:

```
REVOKE SELECT ON invoices FROM sue
```

因为 SUE 的访问权是由 FRANK 授予的。

为了撤销 SUE 的对 INVOICES 表的 SELECT 访问权，要么 FRANK 必须执行 REVOKE 语句，要么使用 WITH GRANT OPTION 授予 FRANK 以 SELECT 访问权的用户 ID（在本例中是 DBOO）必须使用以下两种方法之一：要么使用以下语句撤销 FRANK 的 GRANT OPTION FOR SELECT 选项：

```
REVOKE GRANT OPTION FOR SELECT ON invoices
FROM frank CASCADE
```

要么使用以下语句撤销 FRANK 对 INVOICES 表全部的 SELECT 权限：

```
REVOKE SELECT ON invoices FROM frank CASCADE
```

正如在技巧 108 “使用带 CASCADE 选项的 REVOKE 语句删除权限”中所学习的，撤销对对象的权限或是一项权限的 GRANT OPTION（在 MS-SQL Server 上）也就从原始的 GRANT 语句中所派生的对 INVOICES 表的访问权的所有用户 ID 身上删除了访问权。在本例中，SUE 对 INVOICES 表的 SELECT 权限是从 FRANK 的 GRANT 语句派生出来的。因此，撤销 FRANK 对 INVOICES 表的 SELECT 权限也就取消了 SUE 的派生的 SELECT 权限。

注意：在 DBOO 或另一个授权者使用 REVOKE 语句删除一个用户 ID 的 SELECT 权限之后，如果还有另外的某人向其授予了对对象的同样权限的话，该用户 ID 将仍然有对数据库对象的 SELECT 访问权。例如，如果 WALTER 和 SCOTT 都向 SUE 授予了对 INVOICES 表的 SELECT 权限，而 SCOTT 后来执行了以下语句：

```
REVOKE SELECT ON invoices FROM sue
```

那么 SUE 仍然有对 INVOICES 表的 SELECT 权限，因为 SCOTT 撤销的 SELECT 权限对 WALTER 授予 SUE 的 SELECT 权限没有影响。

技巧 111 理解 MS-SQL Server 对 SELECT 权限的列清单扩展

ANSI/ISO 标准在授予 SELECT 权限时不允许包括列清单。因而，如果一种 DBMS 产品严格地遵循该标准，授予 SELECT 权限就是有还是没有的命题——要么一个用户 ID 具有对表或视图中的所有列的 SELECT 访问权，要么该用户 ID 没有对表或视图中的任何列的 SELECT 访问权。通过对标准的扩展，MS-SQL Server（以及几种其他的 DBMS 产品）允许授权者在想要授予对一个表或视图的某些列的 SELECT 访问权时指定列的清单，而不允许访问列清单之外的列。

例如，假设想要给予利益协调人 PAUL 以访问雇员记录中的 ID、姓名、保险和退休计划信息的权限。但是，并不想让他看到任何的工资、工资等级和定额等数据。MS-SQL Server 允许指定一个用户 ID 使用 SELECT 语句可查询的列，只要在 GRANT 语句中包括列清单，如下所示：

```
GRANT SELECT (id, name, health_plan_selection,
              health_plan_cost, dental_plan_selection,
              dental_plan_cost,
              retirement_plan_participation_pcent,
              retirement_plan_vesting_date)
ON employees TO paul
```

在本例中，如果 PAUL 对 EMPLOYEES 表的他没有查询权的列执行以下 SELECT 语句：

```
SELECT id, name, salary FROM employees
```

MS-SQL Server 将以以下错误信息响应:

```
Server: Msg 230, Level 14, State 1, Line 1
```

```
SELECT permission denied on column 'salary' of object  
'employees', database 'SQLTips', owner 'dbo'.
```

要记住的重要事情是,如果授予 SELECT 权限而没有列清单,就允许用户 ID 看到表或视图内所有列的数据。向 GRANT SELECT 语句中添加列清单就将用户 ID 的 SELECT 权限限制为只显示列出的列中的数据。

注意:如果 DBMS 产品不允许在 GRANT SELECT 语句中指定列清单,仍然可以限制用户只对指定的表列具有 SELECT 访问权,只要对视图而不是其基表本身授予 SELECT 访问权。读者将在技巧 118 “使用视图将 SELECT 权限限制为只对表中的特定列”中学习有关使用视图将用户 ID 的 SELECT 访问权限限制为仅对特定的表列的内容。

技巧 112 使用 GRANT INSERT (以及 REVOKE INSERT) 语句控制对数据库对象的访问

对表或视图的 INSERT 权限允许一个用户 ID 或是角色的成员向表中添加数据行。正如在技巧 111 “理解 MS-SQL Server 对 SELECT 权限的列清单扩展”中所学习的,一个有 SELECT 权限的用户只能“看到”表中的数据而完全不能修改其内容。作为对照,一个只有 INSERT 权限的用户能够改变表的内容(添加行),但却不能查看任何所完成的工作——除非该 ID 还有对对象的 SELECT 权限。另外,如果没有 DELETE 权限(此内容将在技巧 115 “使用 GRANT DELETE[和 REVOKE DELETE]语句控制对数据库对象的访问”中学习),用户就不能删除刚刚添加的行。如果没有 UPDATE 权限(此内容将在技巧 113 “使用 GRANT UPDATE (以及 REVOKE UPDATE) 语句控制对数据库对象的访问”中学习)甚至不能改变行内的数据值。因而,INSERT 权限就是向表中添加行的能力(既可直接添加也可通过视图)。在插入之后,行成为表的一部分,而且添加行的用户 ID 没有对行的附加权限。

用于授予 INSERT 权限的 GRANT 语句的句法如下:

```
GRANT INSERT ON <table name> | <view name>  
TO <user and/or Role name list> [WITH GRANT OPTION]
```

由此,例如,为了允许用户 ID SALLY 对 INVOICES 表执行 INSERT 语句,表的拥有者(或其他具有 GRANT INSERT 权限的用户)必须执行如下的 GRANT 语句:

```
GRANT INSERT ON invoices TO sally
```

或者,为了给予 GARY 以对 PRODUCTS 表的 INSERT 访问权并允许 GARY 向其他用户授予 INSERT 访问权,具有对 PRODUCTS 表的 GRANT INSERT 权限的某人必须在 GRANT 语句中包括 WITH GRANT OPTION,如下所示:

```
GRANT INSERT ON products TO gary WITH GRANT OPTION
```

最后,为了向 LEONARD、MARK 和 SALES_MANAGERS 角色授予对 VW_KEY_CUST_ORDERS 视图的 INSERT 权限,对该视图有 GRANT INSERT 权限的用户必须执行以下 GRANT 语句:

```
GRANT INSERT ON vw_key_cust_orders  
TO leonard, mark, sales_managers
```

注意：为了成功地授予对视图的 INSERT 访问权，视图拥有者（DBOO）必须对视图基于的底层表也具有 GRANT INSERT 访问权。正如读者在技巧 8 “理解视图”中所学习的，视图是虚拟表，因为其中没有任何自己的数据（行）。视图只不过是显示在底层的“真实”表中找到的值而已。由此，当向视图中插入一行时，用户实际上是向视图显示数据的底层表中添加一行。

由于只需要对表有 SELECT 访问权即可创建基于其上的视图，对底层表没有 INSERT 访问权的用户也可创建视图。另外，作为新创建视图的拥有者，该用户可向另一用户授予对视图的 INSERT 访问权（请记住，创建表或视图的用户 ID 具有对其所拥有对象的所有权限——其中包括向另外的用户授予对对象的 INSERT 访问权的 GRANT INSERT 权限）。但是，如果创建了视图的用户没有对其底层表的 INSERT 访问权，DBMS 不允许通过视图向底层表中插入数据。

作为对照，当一个对底层表具有 GRANT INSERT 权限的用户创建一个视图，然后向另外的用户授予对视图的 INSERT 访问权时，接受 INSERT 访问权的用户将能够通过视图向底层表中添加行。例如，给定一个基于 ORDERS 表的 VW_KEY_CUST_ORDERS 视图，对视图和对 ORDERS 表具有 GRANT INSERT 访问权的用户可以执行以下 GRANT 语句：

```
GRANT INSERT ON VW_KEY_CUST_ORDERS TO frank
```

为了向用户 ID FRANK 授予向底层表（ORDERS）插入行的权限，可在 INSERT 语句的 INTO 子句使用该视图的权限，如下所示：

```
INSERT INTO vw_key_cust_orders  
VALUES ('1/1/2001', 1, 8, 6, 258.25, 20)
```

但是，DBMS 不允许 FRANK 使用以下 INSERT 语句直接向 ORDERS 表中添加行：

```
INSERT INTO order VALUES ('1/1/2001', 1, 8, 6, 258.25, 20)
```

因为 FRANK 只有对 VW_KEY_CUST_ORDERS 的 INSERT 访问权，而没有对 ORDERS 本身的 INSERT 访问权。

当想要取消一个用户向表中添加行的能力时，可执行一条 REVOKE INSERT 语句取消以前授予的 INSERT 权限。例如，如果以前向 SALLY 授予了对 INVOICES 表的 INSERT 访问权，提交以下 REVOKE 语句即可取消其向 INVOICES 表中添加行的能力：

```
REVOKE INSERT ON invoices FROM sally
```

在执行了 REVOKE INSERT 语句之后，SALLY 对 INVOICES 表具有的任何其他权限都保留不变。另外，与 SELECT 权限的情况一样，如果 SALLY 接受了另外的用户授予的权限，她仍然有对 INVOICES 表的 INSERT 权限。当撤消一项权限时，DBMS 只取消所授予的权限，而且 REVOKE 语句对另外的用户所授予的对对象的权限没有影响。

技巧 113 使用 GRANT UPDATE（以及 REVOKE UPDATE）语句控制对数据库对象的访问

对表或视图的 UPDATE 权限允许用户或应用程序执行 UPDATE 语句选择性地改变存储在已有行中的数据值。虽然有 UPDATE 权限的用户可改变表的数据，但他们不能添加新行、删除行或是“看到”存储在表中的已有的数据值。

注意：在某些 DBMS 产品（包括 MS-SQL Server）中，用户必须具有对对象的 UPDATE 和 SELECT 两项访问权才能成功地对表或视图执行 UPDATE 语句。例如，在 MS-SQL Server 上执行 UPDATE 语句时，DBMS 首先选择要更新数据的行，然后对所选行执行实际的 UPDATE 语句。由此，如果一个 MSSQL Server 上的用户有对表或视图的 UPDATE 访问权但没有 SELECT

访问权, 则该用户将不能改变表的数据, 因为 DBMS 不允许用户“选择”要用 UPDATE 语句更新的行。因而, 例如, 在 MS-SQL Server 上要授予 UPDATE 访问权, 必须既授予对对象的 UPDATE 访问权也授予 SELECT 访问权。请检查一下系统文档, 看一下, 具体的 DBMS 是否需要用户 ID 具有对对象的这两项访问权才能改变其中的数据值。

用来给予 UPDATE 权限的 GRANT 语句的句法如下:

```
GRANT UPDATE [(column list)]  
ON <table name> | <view name>  
TO <user and/or Role name list> [WITH GRANT OPTION]
```

这样一来, 例如, 为了允许用户 ID KRIS 执行对 INVOICES 表的 UPDATE 语句, 表的拥有者 (或其他有 GRANT UPDATE 权限的用户) 必须执行以下 GRANT 语句:

```
GRANT UPDATE ON invoices TO kris
```

或者, 为了给予 FRANK 以对 PRODUCTS 表的 UPDATE 访问权, 并允许 FRANK 使用 GRANT UPDATE 语句向其他用户授予访问权, 具有对 PRODUCTS 表的 GRANT UPDATE 权限的某用户必须在 GRANT 语句中包括 WITH GRANT OPTION, 如下所示:

```
GRANT UPDATE ON products TO frank WITH GRANT OPTION
```

最后, 为了向 SCOTT、HARMON、LV_STORE MANAGERS 角色 GRANT UPDATE 对 VW_LV_INVENTORY 视图的访问权, 对该视图有 GRANT UPDATE 权限的用户必须执行以下 GRANT 语句:

```
GRANT UPDATE ON vw_lv_inventory  
TO scott, harmon, lv_store_managers
```

注意: 为了成功地授予对视图的 UPDATE 访问权, 视图拥有者 (DBOO) 必须对视图基于的底层表也具有 GRANT UPDATE 访问权。正如读者在技巧 8 “理解视图” 中所学习的, 视图是虚拟表, 因为其中没有任何自己的数据 (行)。视图只不过是显示在底层的“真实”表中找到的值而已。由此, 当更新视图中的数据时, 用户实际上改变的是视图显示数据的底层表中数列的数据。

由于只需要对表有 SELECT 访问权即可创建基于其上的视图, 对底层表没有 UPDATE 访问权的用户也可创建视图。另外, 作为新创建视图的拥有者, 该用户可向另一用户授予对视图的 UPDATE 访问权 (请记住, 创建表或视图的用户 ID 具有对其所拥有对象的所有权限——其中包括向另外的用户授予对对象的 INSERT 访问权的 GRANT UPDATE 权限)。但是, 如果创建了视图的用户没有对其底层表的 UPDATE 访问权, DBMS 不允许通过视图修改底层表中的数据。

作为对照, 当一个对底层表具有 GRANT UPDATE 权限的用户创建一个视图, 然后向另外的用户授予对视图的 INSERT 访问权时, 接受 INSERT 访问权的用户将能够通过视图向底层表中添加行。例如, 给定一个基于 ORDERS 表的 VW_KEY_CUST_ORDERS 视图, 对视图和对 ORDERS 表具有 GRANT UPDATE 访问权的用户可以执行以下 GRANT 语句:

```
GRANT UPDATE ON VW_KEY_CUST_ORDERS TO david
```

为了给予用户 ID DAVID 以更新底层表 (ORDERS) 表数据的权限, 可使用视图作为 UPDATE 语句中的目标表, 如下所示:

```
UPDATE vw_key_cust_orders SET ship_date = '07/11/2001'  
WHERE invoice_date = '07/10/200' AND ship_date IS NULL
```

但是, DBMS 不允许 DAVID 使用类似的 UPDATE 语句直接改变 ORDERS 表中的数据值:

```
UPDATE ORDERS SET ship_date = '07/11/2001'
WHERE invoice_date = '07/10/200' AND ship_date IS NULL
```

以上语句使用了 ORDERS 表而不是视图作为 UPDATE 语句的目标。

DAVID 只有对 VW_KEY_CUST_ORDERS 的 UPDATE 访问权，而没有对 ORDERS 本身的 UPDATE 访问权。

授予对表或视图的 UPDATE 访问权并不是不是有就是无的命题。例如，如果想使用户 ID KAREN 只能够修改 INVENTORY 表中的某些列的数据，而不能修改其他列，可在 GRANT UPDATE 语句中列出 KAREN 可修改的列。例如，以下 GRANT 语句：

```
GRANT UPDATE ON inventory TO karen
```

给予用户 ID KAREN 以改变 INVENTORY 表中任何列中数据的能力，而以下 GRANT 语句：

```
GRANT UPDATE (item_description, item_cost) ON inventory
TO karen
```

只允许 KAREN 修改 INVENTORY 表中的 ITEM_DESCRIPTION 和 ITEM_COST 列。

当想要取消一个用户向表中添加行的能力时，可执行一条 REVOKE UPDATE 语句取消以前授予的 UPDATE 权限。例如，如果以前向 SUE 授予了对 INVOICES 表的 UPDATE 访问权，提交以下 REVOKE 语句即可取消其改变 INVOICES 表中数据的能力：

```
REVOKE UPDATE ON invoices FROM sue
```

在 DBMS 执行了 REVOKE UPDATE 语句之后，用户 ID 对对象具有的任何其他权限都保留不变。另外，与 SELECT 和 INSERT 权限的情况一样，如果用户接受了另外的用户授予的权限，该用户仍然有对对象的 UPDATE 访问权。当撤消一项权限时，DBMS 只取消所授予的权限，而且 REVOKE 语句对另外的用户所授予的对对象的权限没有影响。

正如授予对对象的 UPDATE 权限并不是不是有就是无的命题一样，撤消权限也不是这样。为了只取消对表或视图中特定列的 UPDATE 访问权（与整个对象不同），只要作为 REVOKE 语句的一部分列出不再允许更新的列即可。例如，虽然以下 REVOKE 语句防止用户 ID SUE 更新 INVOICES 表中的任何列：

```
REVOKE UPDATE ON invoices FROM sue
```

但下面的语句：

```
REVOKE UPDATE (invoice_date, cust_id, item_cost)
ON invoices FROM sue
```

只防止用户 ID SUE 更新 INVOICES 表中的 INVOICE_DATE、CUST_ID 和 ITEM_COST 列。

技巧 114 使用 GRANT REFERENCES（以及 REVOKE REFERENCES）语句控制对数据库对象的访问

REFERENCES 权限允许用户引用表的 PRIMARY KEY 和由 UNIQUE 约束所约束的任何其他列作为另一表中的 FOREIGN KEY（读者将在技巧 144“使用 UNIQUE 列约束防止列中的重复值”中学习有关 UNIQUE 约束的内容）。虽然 REFERENCES 权限并不给予用户以直接显示表中数据的能力，但该访问权限添加到 SQL-92 中，以便处理由用户使用 FOREIGN KEY 间接地决定表列中的数据值，而对那些列没有 SELECT 权限所引起的微妙的安全性问题。例如，假设一个用户没有对名为 TAKEOVER_STOCKS_LIST 表的 SELECT 访问权，但却知道 STOCK_SYMBOL 列是该表的 PRIMARY KEY。通过用以下定义创建一个名为

MY_TAKEOVER_STOCKS 的表:

```
CREATE TABLE my_takeover_stocks
(symbol VARCHAR(10)
CONSTRAINT fk_takeover_targets FOREIGN KEY(symbol)
REFERENCES takeover_stocks_list(stock_symbol))
```

结果该用户最终可得到 TAKEOVER_STOCKS_LIST 表中库存的完整清单。

正如读者在技巧 44 “使用 CREATE TABLE 语句指定外键约束”中所学过的, FOREIGN KEY 列的每一行中的值必须存在于由 FOREIGN KEY 约束所引用的表中的列的行之一中。在本例中, 仅当一个库存符号存在于 TAKEOVER_STOCKS_LIST 表中时, DBMS 才允许用户向 MY_TAKEOVER_STOCKS 表中插入该库存符号。由此, 试图向 MY_TAKEOVER_STOCKS 表中插入不在 TAKEOVER_STOCKS_LIST 表中的符号, 将出现以下错误信息从而不能成功:

```
Server: Msg 547, Level 16, State 1, Line 1
INSERT statement conflicted with COLUMN FOREIGN KEY
constraint 'fk_takeover_stocks'. The conflict occurred in
database 'SQLTips', table 'takeover_stocks_list', column
'stock_symbol'.
The statement has been terminated.
```

由于 DBMS 只允许插入已经存在于 TAKEOVER_STOCKS_LIST 表中的库存符号, 在用户试图向 MY_TAKEOVER_STOCKS 表中插入所有可能的库存符号之后, MY_TAKEOVER_STOCKS 表将有 TAKEOVER_STOCKS_LIST 表中的所有库存的清单。

用于授予 REFERENCES 权限的 GRANT 语句的句法如下:

```
GRANT REFERENCES [(column list)] ON <table name>
TO <user and/or Role name list> [WITH GRANT OPTION]
```

这样一来, 为了允许用户 ID ROB 创建一个带有引用了 EMPLOYEES 表中的列的一个或多个 FOREIGN KEY 约束, 表的拥有者 (或其他有 GRANT REFERENCES 权限的用户) 必须执行如下的 GRANT 语句:

```
GRANT REFERENCES ON employees TO rob
```

或者, 为了给予 JAMES 以对 CUSTOMERS 表的 REFERENCES 访问权并允许 JAMES 向其他用户授予 GRANT REFERENCES 权限, 对 CUSTOMERS 表有 GRANT REFERENCES 权限的某人必须在 GRANT 语句中包括 WITH GRANT OPTION, 如下所示:

```
GRANT REFERENCES ON customers TO james WITH GRANT OPTION
```

授予对表的 REFERENCES 权限并不是非有即无的命题。如果想要一个用户仅对表中的某些列可以进行 FOREIGN KEY 引用, 只要作为 GRANT 语句的一部分列出允许的列。例如, 以下 GRANT 语句:

```
GRANT REFERENCES (stock_symbol, security_cussip) ON
takeover_list_table TO karen
```

只允许用户 ID KAREN 使用 STOCK_SYMBOL 和 SECURITY_CUSSIP 列作为她所创建的表中的 FOREIGN KEY 列引用。

注意: 虽然一个表中的 FOREIGN KEY 通常是另一表中的 PRIMARY KEY, 但大多数 DBMS 产品 (包括 MS-SQL Server) 允许使用任何已经施加了 UNIQUE 约束的列作为 FOREIGN KEY 约束引用的列。因而, 如果有一个表, 其中只有唯一一列是 PRIMARY KEY, 在授予

REFERENCES 权限时就不需要指定列名。但是，如果表中有多列都施加了 UNIQUE 约束，可在 GRANT REFERENCES 语句中使用列清单，而且一定要排除其值仍然为对授予 REFERENCES 访问权的用户隐藏的列。

当想要取消一个用户 ID 对表中的列创建 FOREIGN KEY 引用的能力，可执行 REVOKE REFERENCES 语句删除以前授予的权限。例如，如果向 JERRY 授予了对 EMPLOYEES 表的 REFERENCES 访问权，使用以下 REVOKE 语句可取消其引用 EMPLOYEES 表中的列的能力：

```
REVOKE REFERENCES ON employees FROM jerry
```

在 DBMS 执行了 REVOKE REFERENCES 语句之后，任何对对象的其他权限都保留不变。另外，正如 SELECT、INSERT 和 UPDATE 权限一样，如果用户还从其他人处接受了对同一对象的 REFERENCES 访问权，则该用户在执行了 REVOKE REFERENCES 语句之后仍然有对对象的 REFERENCES 访问权。DBMS 只取消所授予的权限。REVOKE 对由另外的用户授予的对对象的相同权限没有影响。

和授予权限一样，撤消对一个对象的 REFERENCES 权限也不是非有即无的命题。为了只 REVOKE REFERENCES 访问表中特定的列，可在 REVOKE 语句中列出用户不再允许引用为 FOREIGN KEY 的列清单。例如，虽然以下 REVOKE 语句：

```
REVOKE REFERENCES ON customers FROM sally
```

防止用户 ID SALLY 引用 CUSTOMERS 表中的任何列，但以下 REVOKE 语句：

```
REVOKE REFERENCES (cust_ID, phone_number) ON customers  
FROM sally
```

只防止引用在 FOREIGN KEY 中引用 CUST_ID 和 PHONE_NUMBER 列。

注意：如果向一个用户授予了对某一对象的 REFERENCES 访问权，而且该用户使用引用对象中的列的 FOREIGN KEY 约束创建了一个表，撤消 REFERENCES 访问权只能防止该用户 ID 定义以后的引用该对象的 FOREIGN KEY。任何在撤消 REFERENCES 权限以前创建的表中的 FOREIGN KEY 引用将继续检查被引用列的数据值，即使在 REFERENCES 权限被撤消之后也是如此。

技巧 115 使用 GRANT DELETE（以及 REVOKE DELETE）语句控制对数据库对象的访问

对表或视图的 DELETE 权限允许用户或应用程序从表中删除一行或多行。正如读者在技巧 112“使用 GRANT INSERT（以及 REVOKE INSERT）语句控制对数据库对象的访问”中所学过的，INSERT 访问权允许用户向表中添加行。但是，在添加了行之后，该用户却没有别的特殊权限。事实上，正如在技巧 110“使用 GRANT SELECT（以及 REVOKE SELECT）语句控制对数据库对象的访问”中所学过的，在没有 SELECT 访问权的情况下，添加行的用户甚至不能要求 DBMS 显示行中的数据值。DELETE 权限与其他每一种访问权限一样，给予用户 ID 以单一的功能——从表中删除行（而不管用户是否添加了行）。

注意：在某些 DBMS 产品（包括 MS-SQL Server）中，用户必须对一个对象既有 DELETE 访问权也要有 SELECT 访问权才能成功地执行如下的 DELETE 语句：

```
DELETE FROM invoices WHERE invoice_date < 01/01/1900
```

例如，MS-SQL Server 上，如果执行语句的用户 ID 对 INVOICES 表有 DELETE 访问权而

没有 SELECT 访问权, 则本例中的 DELETE 语句将不能执行, 会出现以下错误信息:

```
Server: Msg 229, Level 14, State 5, Line 1
SELECT permission denied on object 'invoices', database
'SQLTips', owner 'dbo'.
```

例如, 一个用户只需要 DELETE 访问权即可执行以下 DELETE 语句, 删除雇员表中的所有行:

```
DELETE FROM employees
```

由于从表中删除所有行通常不是执行 DELETE 语句的所期望的结果 (如果有的话), 因而一定要对授予用户 ID 及 DELETE 访问权的对象授予 (GRANT) SELECT 权限。

用于给予 DELETE 权限的 GRANT 语句的句法如下:

```
GRANT DELETE ON <table name> | <view name>
TO <user and/or Role name list> [WITH GRANT OPTION]
```

这样一来, 例如, 为了允许用户 ID JERRY 执行对 INVOICES 表的 DELETE 语句, 表的拥有者 (或其他有 GRANT DELETE 权限的用户) 必须执行以下 GRANT 语句:

```
GRANT DELETE ON invoices TO jerry
```

或者, 为了给予 SCOTT 以对 EMPLOYEES 表的 DELETE 访问权, 并允许 SCOTT 使用 GRANT DELETE 语句向其他用户授予访问权, 具有对 EMPLOYEES 表的 GRANT DELETE 权限的某用户必须在 GRANT 语句中包括 WITH GRANT OPTION, 如下所示:

```
GRANT DELETE ON employees TO scott WITH GRANT OPTION
```

最后, 为了向 SALLY、SUSAN、SHIPPING_RECEIVING_CLERKS 角色 GRANT DELETE 对 VW_SHIPPED_ORDERS 视图的访问权, 具有对该视图有 GRANT DELETE 权限的用户必须执行以下 GRANT 语句:

```
GRANT DELETE ON vw_shipped_orders
TO sally, susan, shipping_receiving_clerks
```

注意: 为了成功地授予对视图的 DELETE 访问权, 授予对视图的 DELETE 访问权的用户必须对视图基于的底层表也具有 GRANT DELETE 访问权。正如读者在技巧 8 “理解视图”中所学习的, 视图是虚拟表, 因为其中没有任何自己的数据 (行)。视图只不过是显示在底层的“真实”表中找到的值而已。因此, 当删除视图中的行时, 用户实际上是从视图显示数据的底层表中删除行。

由于只需要对表有 SELECT 访问权即可创建基于其上的视图, 对底层表没有 DELETE 访问权的用户也可创建视图。另外, 作为新创建视图的拥有者, 该用户可向另一用户授予对视图的 DELETE 访问权 (请记住, 创建表或视图的用户 ID 具有对其所拥有对象的所有权限——其中包括向另外的用户授予对对象的 DELETE 访问权的 GRANT DELETE 权限)。如果创建了视图的用户没有对其底层表的 DELETE 访问权, DBMS 不允许通过视图从底层表中删除行。

作为对照, 当一个对底层表具有 GRANT DELETE 权限的用户创建一个视图, 然后向另外的用户授予对视图的 DELETE 访问权时, 接受 DELETE 访问权的用户将能够通过视图从底层表删除行。例如, 给定一个基于 ORDERS 表的 VW_SHIPPED_ORDERS 视图, 对视图和对 ORDERS 表具有 GRANT DELETE 访问权的用户可以执行以下 GRANT 语句:

```
GRANT DELETE ON vw_shipped_orders TO david
```

为了给予用户 ID DAVID 以从底层表 (ORDERS) 删除值的权限, 可使用视图作为 DELETE 语句中的目标表, 如下所示:

```
DELETE FROM vw_shipped_orders
WHERE shipped_date < '01/01/1999'
```

但是，DBMS 不允许 DAVID 使用如下的类似 DELETE 的语句直接从 ORDERS 表中的删除行：

```
DELETE FROM ORDERS WHERE shipped_date < '01/01/1999'
```

以上语句使用了 ORDERS 表而不是视图作为 DELETE 语句的目标。DAVID 只有对 VW_SHIPPED_ORDERS 的 DELETE 访问权，而没有对 ORDERS 本身的 DELETE 访问权。

当想要取消一个用户从表或视图中删除行的能力时，可执行一条 REVOKE DELETE 语句取消以前授予的 DELETE 权限。例如，如果以前向 WALTER 授予了对 EMPLOYEES 表的 DELETE 访问权，提交以下 REVOKE 语句即可取消从 EMPLOYEES 表中删除行的能力：

```
REVOKE DELETE ON customers FROM walter
```

在 DBMS 执行了 REVOKE DELETE 语句之后，用户 ID 对对象具有的任何其他权限都保留不变。另外，与 SELECT、INSERT、REFERENCES 和 UPDATE 权限的情况一样，如果用户接受了另外的用户授予的权限，该用户仍然有对对象的 DELETE 访问权。当撤消一项权限时，DBMS 只取消所授予的权限，而 REVOKE 语句对另外的用户所授予的对对象的权限没有影响。

技巧 116 使用 GRANT ALL（以及 REVOKE ALL）语句授予（GRANT）或撤消（REVOKE）对数据库对象的权限

在想要授予对一个对象的 SELECT、INSERT、UPDATE、DELETE 和 REFERENCES 访问权时，SQL 提供了一种使用的快捷方式。不用枚举 5 种权限，可简单地执行 GRANT ALL 语句。授予对对象的全部访问权的 GRANT 语句的句法如下：

```
GRANT ALL [PRIVILEGES] [(<column list>)] ON <table name>
TO <user and/or Role name list> [WITH GRANT OPTION]
```

因而，以下 GRANT 语句：

```
GRANT ALL ON employees TO sue
```

与下面的语句是等价的：

```
GRANT SELECT, INSERT, UPDATE, DELETE, REFERENCES
ON employees TO sue
```

为了成功地执行 GRANT ALL 语句，必须有对对象可用的所有安全性权限的 GRANT 选项。在表的情况下，必须授予（GRANT）SELECT、INSERT、UPDATE、DELETE 和 REFERENCES 的访问权。另一方面，如果正在授予对视图的所有访问权限，必须授予（GRANT）对视图及其底层表的 SELECT、INSERT、UPDATE 和 DELETE 访问权，其中没有对视图的 REFERENCE 访问权。

作为数据库对象的拥有者（DBOO），有对所创建的对象的所有权限，因而可成功地对所拥有的对象执行 GRANT ALL 语句。如果某人不是 DBOO，DBOO（或另一位具有全部访问权和用 GRANT 选项的用户）必须与 WITH GRANT OPTION 一起授予（GRANT）某人对对象的 5 种访问权（在视图的情况下是 4 种），这样该人才能向另外的用户 GRANT ALL 对对象的所有权限。

如果提交了一条 GRANT ALL 语句并且缺少一个或多个可用的权限，或者只是向另外的用户授予一种或多种权限，DBMS 将不能执行 GRANT ALL 语句，并且将返回以下错误消息：

```
Server: Msg 4613, Level 16, State 1, Line 1
Grantor does not have GRANT permission.
```

遗憾的是,具体的 DBMS (如在本例中的 MS-SQL Server) 对于缺少哪种访问权或 GRANT 选项可能不明确。

当在 GRANT ALL 语句中指定列清单时一定要小心。由于并不是所有的安全性权限都接受列清单,因而必须检查系统文档,看一下当在 GRANT ALL 语句中出现列清单时,DBMS 会做什么。例如,MS-SQL Server 将对那些接受列清单的权限 (SELECT、UPDATE 和 REFERENCES) GRANT ALL 列清单的权限而对那些不接受列清单的权限 (DELETE 和 INSERT) 忽略列清单。

这样一来,对于 MS-SQL Server 来说,以下 GRANT ALL 语句:

```
GRANT ALL (employee_id, first_name, last_name)
ON employees TO rodger, sue, mary
```

与下面的 GRANT 语句等价:

```
GRANT SELECT, UPDATE, REFERENCES
(employee_id, first_name, last_name)
ON employees TO rodger, sue, mary
GRANT INSERT, DELETE ON employees TO rodger, sue, mary
```

REVOKE ALL 语句撤消在 FROM 子句中列出的用户 ID 和角色对由执行语句的用户在 ON 子句中列出的对象所授予的所有权限。例如,为了删除授予 SUE 对 EMPLOYEES 表的所有权限,可执行以下 REVOKE 语句:

```
REVOKE ALL ON employees FROM sue
```

与 GRANT ALL 语句不同,要执行 REVOKE ALL 语句,不必具有对对象的 WITH GRANT OPTION 的所有访问权。例如,如果只有授予 (GRANT) 对 INVOICES 表的 SELECT、INSERT、UPDATE 和 DELETE 权限而且执行以下 GRANT 语句:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON invoices TO sue
GRANT SELECT, INSERT, DELETE ON invoices TO frank
GRANT SELECT ON invoices TO konrad
```

用以下单条 REVOKE ALL 语句可以撤消所有 (REVOKE ALL) 已经授予 SUE、FRANK 和 KONRAD 的所有权限:

```
REVOKE ALL ON invoices FROM sue, frank, konrad
```

在本例中,已经授予每个用户 ID 以不同的访问权限集,上面的 REVOKE ALL 语句与下面的 REVOKE 语句等价:

```
REVOKE SELECT, INSERT, UPDATE, DELETE ON invoices FROM sue
REVOKE SELECT, INSERT, DELETE ON invoices FROM frank
REVOKE SELECT ON invoices FROM konrad
```

注意:和 REVOKE 语句的情况一样,当执行 REVOKE ALL 语句时,DBMS 只从 FROM 子句中列出的用户 ID 身上取消所授予的对对象的权限。因此,如果你与另一用户授予了用户 ID SUE 以对 CUSTOMERS 表的所有权限,在你执行了以下 REVOKE ALL 语句之后,SUE 仍然有对 CUSTOMERS 表的所有权限:

```
REVOKE ALL ON customers FROM sue
```

一条 REVOKE ALL 语句与一条 REVOKE 语句一样,只取消所授予的权限,而用户仍然保留着从另外人处接受的对对象的所有权限。

技巧 117 使用视图将 INSERT 权限限制为只对表中的特定列

正如读者在技巧 112 “使用 GRANT INSERT（以及 REVOKE INSERT）语句控制对数据库对象的访问”中所学习的，INSERT 权限允许控制谁有向表或视图中添加行的能力。那些具有 INSERT 权限的人允许添加行，而没有此权限的人就不能。SQL-92 通过添加将 INSERT 权限限制为对一个或多个列的能力而扩展了基本的 INSERT 权限，此时不必对行中所有列（或一列也没有）授予 INSERT 访问权。

例如，假设有用以下语句创建的 EMPLOYEES 表：

```
CREATE TABLE employees
(id          INTEGER PRIMARY KEY,
first_name   VARCHAR(25) NOT NULL,
last_name    VARCHAR(30) NOT NULL,
ssan         CHAR(11) NOT NULL,
address      VARCHAR(50),
manager      SMALLINT,
quota        SMALLINT,
hourly_rate  MONEY,
commission_rate MONEY)
```

如果想允许营销部经理 MIKE 添加新雇员，但不想让 MIKE 指定雇员的 MANAGER、QUOTA、HOURLY_RATE 和 COMMISSION_RATE 列。如果 DBMS 支持带列清单的 GRANT INSERT 语句，可用以下语句完成任务：

```
GRANT INSERT (id, first_name, last_name, ssan, address)
ON employees TO mike
```

遗憾的是，并不是所有的 DBMS 产品都允许在 INSERT 语句中使用列清单。如果 DBMS（如 MS-SQL Server）并不支持通过授予对只带某些表列的视图（而不是对表本身）的 INSERT 权限，也仍然可以将 GRANT INSERT 限制为特定的列。

在本例中，可用以下语句创建视图：

```
CREATE VIEW vw_new_marketing_rep_template AS
(SELECT id, first_name, last_name, ssan, address
FROM employees)
```

然后用以下语句授予对视图 VW_NEW_MARKETING_REP_TEMPLATE 的 INSERT 权限：

```
GRANT INSERT ON vw_new_marketing_rep_template TO mike
```

注意：在使用视图将 INSERT 权限限制为只对特定的列时，请记住，当用户使用视图向底层表插入行时，DBMS 对那些不包括在视图中的表列支持 NULL 值。因此，如果视图并不包括所有施加了 NOT NULL 约束的列，用户将不能通过视图向表中插入任何行，即使在授予了对视图的 INSERT 权限之后也不行。

在本例中，当 MIKE 用以下 INSERT 语句添加新雇员时：

```
INSERT INTO vw_new_marketing_rep_template VALUES
(1, 'Konrad', 'King', 'SSAN', '765 E. Eldorado Lane')
```

DBMS 将使用对 ID、FIRST_NAME、LAST_NAME、SSAN 和 ADDRESS 指定的列值向 EMPLOYEES 表中添加行，而且在 MANAGER、QUOTA、HOURLY_RATE 列中放入 NULL 值。如果视图的定义改变为：

```
CREATE VIEW vw_new_marketing_rep_template AS
```

```
(SELECT id, first_name, last_name, address FROM employees)
```

用户 ID MIKE 不能使用视图添加雇员, 因为每个试图向视图 VW_NEW_MARKETING_REP_TEMPLATE 执行的 INSERT 语句, 都会侵犯对 SSAN 列的 NOT NULL 约束。

技巧 118 使用视图将 SELECT 权限限制为只对表中的特定列

技巧 110 “使用 GRANT SELECT (以及 REVOKE SELECT) 语句控制对数据库对象的访问”向读者展示了如何使用 GRANT SELECT 语句允许一个用户显示表或视图中的列值。与 INSERT、UPDATE 和 REFERENCES 权限不同, SQL-92 标准不允许 SELECT 语句中的列清单。因此, 如果具体的 DBMS 完全按该标准定义实现 SELECT 语句, 则必须允许一个用户 ID 要么看到表中所有的列, 要么什么也看不到。幸运的是, 正如 GRANT INSERT 语句的情况一样, 可以绕过 SELECT 语句的“无列清单”限制, 只要授予对视图而不是对底层表本身的 SELECT 权限即可。

注意: 某些 DBMS, 包括 MS-SQL Server, 扩展了标准的 SELECT 语句, 使之允许列清单(正如在技巧 110 中所学习的)。请检查一下系统手册中的 GRANT 句法, 找出具体的 DBMS 中支持列清单的权限。

为了允许一个用户只看到表中的某些列, 只用那些想要让用户看到的列来创建视图, 并对视图而不是底层表授予 (GRANT) SELECT 权限。例如给定在技巧 117 “使用视图将 INSERT 权限限制为只对表中的特定列”中定义的 EMPLOYEES 表, 可使用以下 CREATE 语句创建视图 VW_MARKETING_REPS:

```
CREATE VIEW vw_marketing_reps AS
(SELECT id, first_name, last_name, ssan, address, manager,
      quota FROM employees)
```

然后使用以下语句向 MARKETING_EMPLOYEES 角色 GRANT SELECT 对 VW_MARKETING_REPS 的 SELECT 权限:

```
GRANT SELECT ON vw_marketing_reps TO marketing_employees
```

MARKETING_EMPLOYEES 角色的成员接着可执行 SELECT 语句来显示从 EMPLOYEES 表获得全部数据的 VW_MARKETING_REPS 视图中的任何 (或全部) 列。由于两个支付列 (HOURLY_RATE 和 COMMISSION_RATE) 在视图列清单中忽略了, 用户就不能显示这两列中的数据。事实上, 人们甚至不知道还有这两列存在。

技巧 119 使用视图扩展 SQL 安全性权限

正如读者在技巧 117 “使用视图将 INSERT 权限限制为只对表中的特定列”和技巧 118 “使用视图将 SELECT 权限限制为只对表中的特定列”中所学过的, 可使用视图来限制用户可使用 SELECT 语句显示表中的哪些列, 以及在使用 INSERT 语句向表中添加行时, 可向哪些列中放入数据。这两个技巧都展示了使用视图来限制用户可看到和修改的列的方法。还可以使用视图来限制用户对表内特定行的访问 (除了对表中的指定列之外)。

例如, 假设有几个销售办公室, 每个办公室都允许管理在集中的 EMPLOYEES 表中的本办公室内雇员的记录。在 GRANT UPDATE 语句中使用列清单, 就可限制经理对特定列做出改变的能力, 如下所示:

```
GRANT UPDATE (first_name, last_name, address)
```

```
ON employees TO sales_office_managers
```

但是，通过授予对包括所有办公室内雇员信息的 EMPLOYEES 的 UPDATE 访问权，一个办公室的经理可改变工作于不同办公室的雇员的数据。通过授予对视图（而不是底层表）的相同的 UPDATE 权限，就可限制 UPDATE 权限，使经理只能改变工作于自己办公室内的雇员的数据。

例如，如果用如下语句为每个办公室创建视图：

```
CREATE VIEW vw_officel_employees AS
(SELECT * FROM employees WHERE office = 1)
CREATE VIEW vw_office2_employees AS
(SELECT * FROM employees WHERE office = 2)
```

就可使用以下 GRANT 语句：

```
GRANT UPDATE (first_name, last_name, address)
ON vw_officel_employees TO officel_managers
GRANT UPDATE (first_name, last_name, address)
ON vw_office2_employees TO office2_managers
```

限制每个办公室的经理只能更新其自己雇员的个人信息。

在本例中创建的视图还可说明将 SELECT 访问权限制为只对表内的某些列。通过使用以下 GRANT 语句授予权限：

```
GRANT SELECT ON vw_officel_employees TO officel_managers
GRANT SELECT ON vw_office2_employees TO office2_managers
```

就可限制 OFFICE1_MANAGERS 角色只能显示工作于办公室 1 中的雇员的表列，而 OFFICE2_MANAGERS 的角色成员只能看到办公室 2 中雇员的信息。

注意：为了进一步将 SELECT 访问权限制到特定表行的特定列，可修改 CREATE VIEW 语句，使之只选择想要让用户看到的行和列。例如，以下 CREATE VIEW 语句：

```
CREATE VIEW vw_officel_employees AS
(SELECT first_name, last_name, address
FROM employees WHERE office = 1)
```

在执行了本技巧前面的 GRANT UPDATE 和 GRANT SELECT 语句之后，将只允许 OFFICE1_MANAGERS 角色的成员显示并修改办公室 1 中的雇员的 FIRST_NAME、LAST_NAME 和 ADDRESS 列。

类似于限制 SELECT 和 UPDATE 权限，还可使用视图将 DELETE 权限限制为只对表中特定的行。例如，假设想要允许发货部经理（用户 ID 为 FRANK）删除任何退回的多于 6 个月的定单。可使用以下语句创建老的退回定单的视图：

```
CREATE VIEW vw_backorders_180 AS
(SELECT * FROM orders WHERE date_shipped IS NULL AND
(GETDATE() - order_date) > 180)
```

并使用以下语句授予 FRANK 以只对那些行的 DELETE 权限：

```
GRANT DELETE ON vw_backorders_180 TO frank
```

用户 ID FRANK 接着就可使用以下语句从系统中删除退回的老的定单：

```
DELETE FROM vw_backorders_180
```

注意：用在本例中的 GETDATE() 是 MS-SQL Server 内建的函数，可以 DATETIME 格式返回当前的系统日期和时间。

第 7 章 创建索引加快数据引用

技巧 120 理解 MS-SQL Server 如何选择用于查询的索引

MS-SQL Server 具有在单个查询中每个表使用多个索引的能力。由此，MS-SQL Server 7.0 版的多个搜索条件查询执行是对以前版本的极大改进，因为通过创建基于正确表列的索引，用户实际上消除了对表的扫描（DBMS 在此过程中要读取表中的每一行）。在表上创建一个或多个索引之后，通常应该让 MS-SQL Server 的查询优化器来决定在查询执行期间使用哪个索引。

为了使 MS-SQL Server 使用索引，SELECT 语句的 WHERE 子句中的列之一必须是索引中的第一列。例如，如果有使用以下语句创建的 CALL_HISTORY 表：

```
CREATE call_history
(phone_number INTEGER,
date_called DATETIME,
call_time SMALLINT,
hangup_time SMALLINT,
disposition VARCHAR(4),
called_by CHAR(3))
```

并用以下语句创建索引：

```
CREATE INDEX date_index
ON call_history (date_called, call_time, phone_number)
```

MS-SQL Server 在执行以下查询时绝不会选择 DATE_INDEX 索引：

```
SELECT * FROM call_history WHERE phone_number = 2631070
```

因为在搜索条件（PHONE_NUMBER）中指定的该列不是索引中的第一列。

注意：读者将在技巧 121 “使用 CREATE INDEX 语句创建索引”中学习如何使用 CREATE INDEX 语句创建索引。现在，要了解的重要事情是，MS-SQL Server 仅当索引的第一列是 SELECT 语句的 WHERE 子句中指定的列之一时才选择使用索引。

作为对照，如果通过以下 CREATE 语句创建了另一索引：

```
CREATE INDEX caller_index
ON call_history (called_by, date_called)
```

并提交了以下查询：

```
SELECT * FROM call_history
WHERE called_by = 'RRH' AND phone_number = 2631056
```

MS-SQL Server 的优化器将检查总结正在查询的表中的信息分配的系统表，并确定扫描表或进行索引查询哪一个提取 SELECT 语句所需数据的最好方法。如果 DBMS 确定索引查询是最好的，它将使用第二个索引（CALLER_INDEX）作为所选的惟一可用的索引（在本例中），因为其第一列（CALLED_BY）是用在 SELECT 语句搜索条件中的列之一。

当执行一条查询时，不用让 MS-SQL Server 的优化器来选择最好的索引，可通过在 SELECT 语句的 FROM 子句中包括 INDEX = 子句告诉 DBMS 使用哪个索引。例如，以下

SELECT 语句:

```
SELECT * FROM call_history INDEX=caller_index  
WHERE date_called BETWEEN '01/01/2000' AND '01/31/2000'
```

强制 DBMS 使用 CALLER_INDEX, 否则 MS-SQL Server 可能会选择 DATE_INDEX 以优化查询。

注意: 当强制 DBMS 使用一种特定的索引而不是让其优化器选择最好的索引时一定要十分小心。在本例中, 强制使用 CALLER_INDEX 索引可能引起很大且不必要的开销, 因为 DBMS 必须读取索引中的每条条目, 以满足查询。如果 MS-SQL Server 被允许使用 DATE_INDEX, 则 DBMS 就会使用有效的算法来找出满足搜索标准的索引条目 (DATE_CALLED BETWEEN '01/01/200' AND '01/31/2000'), 而且只要遇到第一个不在可接受的日期范围内的日期时, 就会停止读取索引中的行。

技巧 121 使用 CREATE INDEX 语句创建索引

对于大型表来说, 执行全表扫描找到列值满足查询的搜索标准的行可能是非常费处理时间的。正如使用系统手册的字母顺序的索引一样可以帮助读者快速找到所需要的信息, 在数据库索引中通过排序值可以让 DBMS 以最小的系统开销找到并提取满足查询条件的表行。

在执行 CREATE INDEX 语句时, DBMS 一次一行地读取要创建索引的表。在读取每一行时, 系统创建一个键字 (根据被索引的列的串接), 并将键字与指向产生键字的行的物理磁盘位置一起插入索引。

CREATE INDEX 的基本句法如下:

```
CREATE [UNIQUE] INDEX <index name> ON <table name>  
(<column name> [ASC | DESC][, ..., <last column name>  
[ASC | DESC]])
```

因而, 为了创建 INVOICES 表的基于 INV_DATE 和 INV_NO 列的索引, 可使用以下 CREATE INDEX 语句:

```
CREATE INDEX date_index ON INVOICES (inv_date, inv_no)
```

注意: 某些 DBMS 产品允许指定索引中的值是以升序 (ASC) 或是以降序 (DESC) 排序。当在 CREATE INDEX 语句中使用 ASC 或 DESC 时, 请检查一下系统手册, 确保具体的 DBMS 产品支持索引的排序选项。例如, MS-SQL Server 允许在 CREATE INDEX 语句的列名后添加 ASC 或 DESC, 但其忽略这两种属性并总是将索引以升序排序。

向 CREATE INDEX 语句中添加关键词 UNIQUE 就告诉 DBMS 创建一个只允许非重复条目的“惟一的”索引。在本例中, 如果将 CREATE INDEX 语句改变为:

```
CREATE UNIQUE INDEX date_index  
ON INVOICES (inv_date, inv_no)
```

仅当在 INVOICES 表中的每一行中 INV_DATE 和 INV_NO 列中的数据对具有惟一值时 DBMS 才创建索引 DATE_INDEX——也就是说, 在表中可能只有一行有 INV_DATE 值为 02/01/2000 而且 INV_NO 值为 5。另外, 在 DBMS 创建了惟一索引之后, 如果添加行键字值需要 DBMS 向惟一索引中插入一个重复的条目, 则将一行插入已索引的表的企图将不能成功而出现错误信息如下:

```
Server: Msg 2601, Level 14, State 3, Line 1
```



```
Cannot insert duplicate key row in object 'invoices' with  
unique index 'date_index'.  
The statement has been terminated.
```

注意：当创建惟一索引时，应确保没有被索引的列允许 NULL 值。例如，MS-SQL Server 将 NULL 看作是不同的值，因而，如果创建了单列的惟一索引，MS-SQL Server 将只允许用户向表中添加用 NULL 值填入索引列的一行（添加第二行就要求在惟一索引中的一条重复的键字条目[两个 NULL]）。

技巧 122 理解 MS-SQL Server 的 CREATE INDEX 语句选项

技巧 121 “使用 CREATE INDEX 语句创建索引”引用了以下的句法作为 CREATE INDEX 语句的基本句法：

```
CREATE [UNIQUE] INDEX <index name> ON <table name>  
    (<column name> [ASC | DESC] [,...<last column name>  
    [ASC | DESC]])
```

因为每种 DBMS 产品提供其自己的附加选项用于告诉系统诸如创建索引时使用的硬盘驱动器或位置、要有多少自由空间留给每个索引页上的附加数据以及索引如何维护之类的信息。

例如，在 MS-SQL Server 上 CREATE INDEX 语句的句法如下：

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED]  
INDEX <index name> ON <table name>  
    (<column name> [,...<last column name>])  
[WITH [DROP_EXISTING]  
    [{,} FILLFACTOR = <fill factor>]  
    [{,} PAD_INDEX]  
    [{,} IGNORE_DUP_KEY]  
    [{,} STATISTICS_NORECOMPUTE] ]  
[ON <filegroup name>]
```

本技巧的其余部分将解释每个在 MS-SQL Server 上创建索引时可用选项的意义。如果使用的不是 MS-SQL Server，请自己熟悉一下每个选项的目的，然后检查一下系统手册，看有什么可用的选项。

● UNIQUE

在技巧 121 中，读者已经学习了，如果行中的列值生成了已经存在于索引表的惟一索引中的键字值时，对索引应用 UNIQUE 选项可防止向索引后的表内插入行。实际上，基于两个或多个列创建索引并指定 UNIQUE 选项是将非 PRIMARY KEY 的多列 UNIQUE 约束应用于要索引的表的一种方法。由于索引的列值的每种组合必须在索引中是惟一的，然后同样的列值组合必须在本身之内也是惟一的。向表内插入行并没有向表上的每个索引行添加键字值。

● CLUSTERED/NONCLUSTERED

MS-SQL Server 的非集群索引的作用与对索引所期望的一样。DBMS 在非集群索引中对键字值排序（以升序），而索引的表行不排序。作为对照，当创建一个集群索引时，MS-SQL Server 仅对索引中的键字值排序，而且对表中的行排序，以便使其与索引的排序相匹配。读者将技巧 125 “理解 MS-SQL Server 的集群索引”中学习有关集群索引的更多的内容。在默认情况下，如果在创建索引时既没有指定集群也没有指定非集群选项，MS-SQL Server 创建一个非集群索引。

● DROP_EXISTING

由于表只可有一个集群索引，如果想要重新创建索引或是基于不同的索引列的集合或顺序创建新索引，则首先必须将已有的集群索引删除(DROP)。无论何时删除一个集群索引，MS-SQL Server 自动地重新建立表的每一个留下的非集群索引。只要为表创建集群索引，服务器还重新创建表的所有非集群索引。这样一来，删除一个集群索引和使用 DROP INDEX 语句，然后再使用 CREATE CLUSTERED INDEX 语句重新创建可使 MS-SQL Server 建立每个非集群索引两次。

DROP_EXISTING 选项允许使用单条语句重新创建一个集群索引。当 DBMS 执行一条带有 DROP_EXISTING 选项的 CREATE CLUSTERED INDEX 语句时，删除已有的集群索引，重新创建，然后重建所有的非集群索引。

● FILLFACTOR

MS-SQL Sever 将表和索引数据存储在 2KB 的页面上。索引页以树形排列链接在一起，如图 7.1 所示。

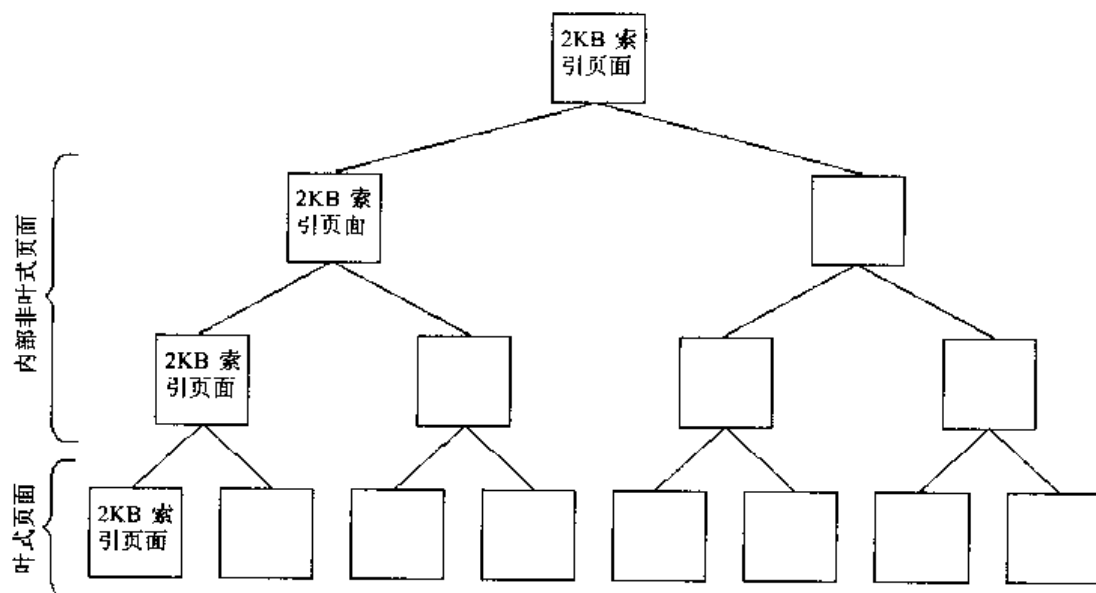


图 7.1 MS-SQL Server 索引页树

MS-SQL Server 使用二进制树在索引中查找特定的键字值的方法超出了本书的范围。要了解的重要事情是，索引键字值存储在 2KB 页面上，而且只要索引页变满，DBMS 必须将页面分成两个，并添加附加的节点，将空间用于索引页面上的更多的键字值。

页面分割对系统开销来说是很大的，因而应该尽可能避免。FILLFACTOR 选项告诉 DBMS 在执行 CREATE INDEX 语句时，为每个所创建的叶片页面上的附加键字值留下多少空间。

注意：不管为 FILLFACTOR 提供的值为多少，对于集群索引，DBMS 总是创建有每一附加条目所需空间的内部（非叶式）页面，而为非集群索引创建有两个附加条目所需空间的内部页面。

例如，FILLFACTOR 为 100 时，告诉 DBMS 将每个叶式页面中的 2KB 的 100%可用空间用于已有的索引值。虽然完全充满的叶式页面创建最小可能的索引，但这样做时，如果向表中添加了行或是索引的列值发生了变化，也产生大量的索引页面分割。当 FILLFACTOR 为 100 时，没有叶式页面有保存附加键字值的空间，而且行插入或索引列值的变化将引起页面的分割。

因而应该只对“只读”的表（将来不会插入行和更新数据）指定 100% 的 FILLFACTOR 选项值（FILLFACTOR=100）。

为那些还没有包括其完整的数据集的表索引，可使用较小的 FILLFACTOR 值，如 10%。例如，在指定 FILLFACTOR 值为 10% 时，DBMS 将充满叶式页面 10% 的容量。因而，如果组成索引的列尺寸总和是 200 个键字可添入 2KB 的页面，那么 DBMS 将使每个叶式页中有 20 个键字值——而留下其余的 180（90%）用于保存附加的键字，当向索引的表中添加新行或是更新索引列值时，就不需索引页分割。

如果并未明确指定 FILLFACTOR（1~100）的值，MS-SQL Server 将其设置为 100 并填满每个叶的 100% 的容量。

- PAD_INDEX

仅当指定了非 0 的 FILLFACTOR 选项时 PAD_INDEX 才有用。虽然 FILLFACTOR 告诉 DBMS 当建立新索引时，使用 2KB 叶式页面空间中的百分之多少，而 PAD_INDEX 选项告诉 DBMS 在索引树中将 FILLFACTOR 的使用百分数用于非叶式的内部页。这样一来，如果指定 FILLFACTOR=10，而且还在 CREATE INDEX 语句中添加了 PAD_INDEX 选项，则 DBMS 将生成只包括大约 205 字节数据（2KB 的 10%）的叶式和非叶式索引。

不管指定 FILLFACTOR 的值为多少，DBMS 总是在每个内部页面上放入至少一个键字值，而为至少一个附加关键字条目留下空间（对非集群索引来说是两个条目）。

- IGNORE_DUP_KEY

指定 IGNORE_DUP_KEY 选项，如果表已经在列（或列组合）中包括了重复的值，就不允许创建一列的惟一索引。IGNORE_DUP_KEY 只影响 MS-SQL Server 处理将来使用 UPDATE 和 INSERT 语句试图向惟一索引中添加有重复的键字值的行时的方式。

不管是否包括了 IGNORE_DUP_KEY 选项，如果列值会引起向惟一索引中添加重复键字，系统都将不允许向表中添加行。但是，如果没有 IGNORE_DUP_KEY 选项，DBMS 将以错误消息中止重复的键字插入，并返回所有的事务处理过程至该点。作为对照，如果在 CREATE UNIQUE INDEX 语句中包括了 IGNORE_DUP_KEY 选项，MS-SQL Server 仍然中止重复的键字插入企图，但却发出警告信息（而不是产生错误）并继续事务处理过程中的下一条语句。

- STATISTICS_NONRECOMPUTE

在创建索引期间，MS-SQL Server 在特殊的统计页记下有关索引列中的数据值的分布。DBMS 以后在决定以最小时间回答查询使用哪个索引时可使用该统计页。向 CREATE INDEX 语句中添加 STATISTICS_NONRECOMPUTE 选项可告诉 MS-SQL Server 在统计数字由于行插入、删除和索引列值的变化变得过时，不要自动地周期性地重新计算索引统计。

设置 STATISTICS_NONRECOMPUTE 选项可消除在执行周期性的表扫描以更新索引的统计页时的开销。但是，在查询时，过时的统计会妨碍查询优化器选择最佳索引。使用“错误的”索引会降低 DBMS 返回查询结果的速度，事实上，为查询结果强制扫描而不是允许有效的索引搜索抵消了开销的减少。

- ON <filegroup name>

FileGroup 参数是 MS-SQL Server 用来引用数据库可用来保存其对象的物理磁盘空间的逻辑名。当执行 CREATE DATABASE 语句时，MS-SQL Server 创建 FileGroup PRIMARY 并在分配给 PRIMARY FileGroup 的磁盘空间内存储所有表、视图、索引、存储过程等。可在任何时

间创建并向数据库中添加 FileGroups（命名的物理磁盘区域）。

在数据库中有多个 FileGroups 的好处是，可将其分布在多个物理磁盘驱动器上，这就允许系统硬件对数据库对象同时执行 I/O 操作。例如，假设让 DBMS 在 C 驱动器上创建 PRIMARY FileGroup，然后在 D 驱动器上创建 FileGroup FILEGROUP2_D，再在 E 驱动器上创建 FILEGROUP3_E。以下 CREATE INDEX 语句将把两个索引中的每一个放在不同的磁盘器上：

```
CREATE INDEX product_index ON invoices
    (product_code, inv_date, inv_no) ON FILEGROUP2_D
CREATE INDEX date_index ON invoices
    (inv_date, inv_no) ON FILEGROUP3_E
```

因此，当向 INVOICES 表中插入新行时，DBMS 可告诉系统硬件来更新 C 驱动器上的表，然后让其同时更新两个索引。如果两个索引位于同一个 FileGroup 上，操作系统必须先更新一个索引然后再更新另一个索引。

技巧 123 使用 MS-SQL Server Enterprise Manager 创建索引

除了 CREATE INDEX 语句（读者已经在技巧 121 “使用 CREATE INDEX 语句创建索引”和技巧 122 “理解 MS-SQL Server 的 CREATE INDEX 语句选项”中学习了有关内容）之外，MS-SQL Server 还提供了图形方法来通过 Enterprise Manager 创建索引。

为了使用 MS-SQL Server Enterprise Manager 创建索引，请执行以下步骤：

1. 为了启动 Enterprise Manager，可单击 Start 按钮，将鼠标指针移动到 Start 菜单的 Programs 上并选择 Microsoft SQL Server 7.0，然后单击 Enterprise Manager。
2. 为了显示 SQL 服务器的列表，可单击 SQL Server Group 左边的加号 (+)。
3. 单击想要在其上创建索引的 SQL Server 左边的加号 (+)。例如，如果想要在名为 NVBizNet2 的 SQL Server 上创建索引，可单击 NVBizNet2 图标左边的加号 (+)。Enterprise Manager 将显示包括在所选择的 MS-SQL Server 上可用的数据库和服务的文件夹。
4. 单击 Databases 文件夹左边的加号 (+)。Enterprise Manager 将显示在步骤 3 中所选的服务器管理的数据库清单。
5. 选择包括想要在其上创建索引的表的数据库。对本例来说，在展开的 Databases 清单中单击 SQLTips 图标左边的加号 (+)，Enterprise Manager 将显示 SQLTips 数据库中的数据库对象类型的图标。
6. 单击 Tables 图标。Enterprise Manager 将在其右窗格中显示 SQLTips 数据库中表的清单。
7. 单击想要创建索引的表（Enterprise Manager 对表的清单以字母顺序排序。如果看不到所要的表，可使用窗口右边的滚动条显示其余的数据库表）。对本例来说，单击 INVOICES。
8. 选择 Action 菜单上的 All Tasks（所有任务）选项并单击 Manage Indexes（管理索引）。Enterprise Manager 将显示 Manage Index 对话框。
9. 为了定义新索引，单击 New 按钮。Enterprise Manager 将显示 Create New Index（创建新索引）对话框，如图 7.2 所示。
10. 在 Name 域中键入索引名称。虽然并不是任何合法的对象名都允许作为索引名，但使用索引中的第一列的名称作为名称的任一部分或开头，可使之易于看出 MS-SQL Server 是否使用了索引来加速查询（在技巧 120 “理解 MS-SQL Server 如何选择用于查询的索引”中，读

者已经了解, MS-SQL Server 仅当用在 SELECT 语句搜索标准中的列是索引的第一列时才将该索引用于查询)。另外, 如果在对 FOREIGN KEY 列的索引前面放上 fk_ 而在 PRIMARY KEY 的索引前面放上 pk_, 则易于跟踪哪个关键字已经索引哪个没有索引。对本例来说, 在 Index Name 域中键入 product_code_index。



图 7.2 MS-SQL Server Enterprise Manager 的 Create New Index 对话框

11. 为了选择索引的列, 可单击想要包括的每个列名左边的复选框。对本例来说, 单击 PRODUCT_CODE 和 INV_DATE 左边的复选框, 使之出现选择标记。

12. 使用 Move Up (向上移动) 和 Move Down (向下移动) 按钮指定索引中列的顺序。为达此目的, 单击 INV_DATE, 然后重复单击 Move Down 按钮直到 INV_DATE 出现在对话框中列清单部分中的 PRODUCT_CODE 之后。

13. 单击对话框的 Index 选项区域中想要选择的每个索引选项左边的复选框为索引选择附加的选项 (在技巧 122 中已经学习了每个选项的意义)。对本例来说, 使所有索引选项的复选框中空着。

14. 为了创建索引并返回 Manage Index 对话框, 单击 OK 按钮。

15. 对每个想要创建的新索引重复步骤 9~步骤 14。如果需要改变索引的定义, 可单击想要修改的索引, 然后单击 Edit 按钮。或者, 为了删除 (DROP) 索引, 单击 Delete 按钮。

16. 当完成为表定义索引时, 单击关闭按钮。

正如读者在技巧 120 中所学习的, 索引中列的顺序是非常重要的。因而, 一定要在步骤 12 中将索引中的列正确地排列。正确的列和错误顺序的索引是没有用处的。例如, 使 INV_DATE 仍然为 PRODUCT_CODE_INDEX 中的第一列将使 DBMS 必须执行全表扫描才能满足以下查询:

```
SELECT SUM(qty) FROM invoices WHERE product_code = 4
```

而重新排列索引列, 使 PRODUCT_CODE 在先而 INV_DATE 在后将使 MS-SQL Server 使

用 `PRODUCT_CODE_INDEX` 索引来减少提取表行的数目，只要提取产品代码为 4 的行即可。

技巧 124 使用 DROP INDEX 语句删除索引

虽然表索引通过减少所需的读操作数目可大大地减少 DBMS 返回查询结果所花时间，但索引却占用磁盘空间，而且如果索引表经常更新的话，对性能有负面影响。请记住，更新索引中的列值所花的处理开销是更新不在索引中的同样列值的两倍。

当改变索引列的值时，DBMS 不仅必须改变表中的列值，而且还必须更新包括列的每个索引。由于 DBMS 将索引作为附加的表来存储，更新是 3 个索引的一部分的列引起 4 个表的更新，而对没有索引的表只更新一个表即可。类似地，每个 `DELETE` 语句不仅必须从主表中删除一行，而且还必须从该表的每个索引中删除一行。另外，`INSERT` 语句要增加更多的开销，因为每次都要求 DBMS 向索引表中和表的每个索引中添加数据（除非更新的是出现在该表的每个索引中的列，这时 DBMS 在存储每个更新的索引列值时，不必改变每个索引）。

为了使数据库索引的开销和磁盘空间使用率最小化，可使用 `DROP INDEX` 语句删除不再需要的索引。与 `CREATE INDEX` 语句类似，`DROP INDEX` 语句的句法随不同的数据库产品而有所不同。某些只需要索引名，如：

```
DROP INDEX <index name>
```

而其他产品要求表名和索引名，如：

```
DROP INDEX <table name>.<index name>
```

例如，MS-SQL Server 要求表名和索引名。因而，要从 `INVOICES` 表中删除 `PRODUCT_CODE_INDEX` 索引，可使用以下 `DROP INDEX` 语句：

```
DROP INDEX invoices.product_code_index
```

技巧 125 理解 MS-SQL Server 的集群索引

集群索引是 MS-SQL Server 表的特殊索引，可强制 DBMS 以索引的准确顺序来存储表数据。使用集群索引的好处有如下 3 点：

- 表将使用所需的最小磁盘空间，因为 DBMS 在插入新行时会自动地重用以前分配给删除行的空间。
- 带有基于集群索引的列值范围条件的查询执行得更快，因为范围内的所有值在物理磁盘上都互相靠近。
- 那种基于集群索引中的列以升序显示数据的查询不需要 `ORDERED BY` 子句，因为表数据已经以所要求的输出顺序排列。

当创建集群索引时，请记住，表可有一个且只能有一个这样的索引。毕竟，表行必须以集群索引的顺序排列，而且单个表在磁盘上只能有一个物理记录排列方式。

为了创建集群索引，可在 `CREATE INDEX` 语句（读者在技巧 121 “使用 `CREATE INDEX` 语句创建索引”和技巧 122 “理解 MS-SQL Server 的 `CREATE INDEX` 语句选项”中学习的内容）中添加关键词 `CLUSTERED`。例如，为了创建基于 `INVOICES` 表的 `PRODUCT_CODE` 和 `INV_DATE` 列的集群索引，可执行以下 `CREATE INDEX` 语句：

```
CREATE CLUSTERED INDEX cl_product_code_index  
ON invoices (product_code, inv_date)
```

注意：索引名 CL_PRODUCT_CODE_INDEX 前的 CL_ 是可选的。但是，如果用 CL_ 开始每个集群索引名，即可容易地区分特定表的集群索引与非集群索引。

如果为一个包括数据的表定义了集群索引，MS-SQL Server 在创建索引时将锁定该表。因而，在为包括大量行的表创建集群索引时，请选择创建时间，确保表数据对用户和应用程序不可用时对大家来说没有什么不便。在建立索引时，DBMS 不仅向索引中插入集群索引列值，而且还重建表本身，使行以与出现在索引中的键字条目的同样顺序排列。

在对表创建了一个集群索引之后，DBMS 将自动地重新安排行，在插入新行或是更新是集群索引一部分的列值时，使之以与出现在索引中的键字条目的同样顺序排列。由此，对那些经常大量插入行或更新集群索引列值的表创建集群索引并不是好主意。在物理上移动表行以及更新索引所涉及的开销将快速地抵消从忽略 ORDER BY 子句以及快速执行值范围搜索中获得的性能提高。

注意：当执行既没有 CLUSTERED 也没有 NONCLUSTERED 关键词的 CREATE INDEX 语句时，DBMS 将创建 NONCLUSTERED 索引。

技巧 126 使用 MS-SQL Server Index Tuning Wizard (索引调节向导) 优化数据库索引

在技巧 120~技巧 125 中，读者已经了解，大多数 DBMS 产品都支持索引，因为索引通过加快查询执行速度和减少由不必要的全表扫描引起的开销，可大大地改善整体系统性能。但是，读者也知道，选择了错误的列来索引，实际上会对系统响应时间造成负面影响。如果大多数系统的查询是基于未索引的列，而那些索引了的列又很少出现在 SELECT 语句的 WHERE 子句中，DBMS 不仅不能最大化索引的好处，而且还招致附加的维护无用索引的开销。

MS-SQL Server 提供 Server Profiler (服务器概况分析器) 和 Index Tuning Wizard (索引调节向导) 来帮助用户创建自己的操作环境中优化的一套索引。在 Server Profiler 捕捉到对数据库提交的跟踪文件的所有查询的日志时，Index Tuning Wizard 可分析日志中的语句，以便确定能够生成可大大地改善查询和整体 DBMS 性能的索引。使用列出系统查询的代表性示例的跟踪文件，Index Tuning Wizard 的建议是根据服务器的实际工作负载相对于某些理论数据库模型而作出的。

为了使用 MS-SQL Server 的 Index Tuning Wizard，可执行以下步骤：

1. 为了启动 Enterprise Manager，可单击 Start 按钮，将鼠标指针移动到 Start 菜单的 Programs 上并选择 Microsoft SQL Server 7.0，然后单击 Enterprise Manager。
2. 为了显示 SQL 服务器的列表，可单击 SQL Server Group 左边的加号 (+)。
3. 单击服务器图标，选择管理着包括要在其上运行 Index Tuning Wizard 的表的数据库的 SQL 服务器。例如，如果想要在名为 NVBizNet2 的 SQL Server 上运行 Index Tuning Wizard，可单击 NVBizNet2 图标。
4. 选择 Tools 菜单上的 Wizards 选项。Enterprise Manager 将显示 Select Wizard (选择向导) 对话框。
5. 为了显示数据库管理向导的清单，可单击 Management (管理) 左边的加号 (+)。
6. 在展开的数据库管理向导清单上单击 Index Tuning Wizard 条目，然后单击 OK 按钮。Enterprise Manager 将启动 Index Tuning Wizard，其中显示其 Welcome to the Index Tuning Wizard (欢迎使用索引调节向导) 消息框。
7. 单击 Next 按钮前进到 Index Tuning Wizard-Select Server and Database (索引调节向导—

选择服务器和数据库）对话框，如图 7.3 所示。

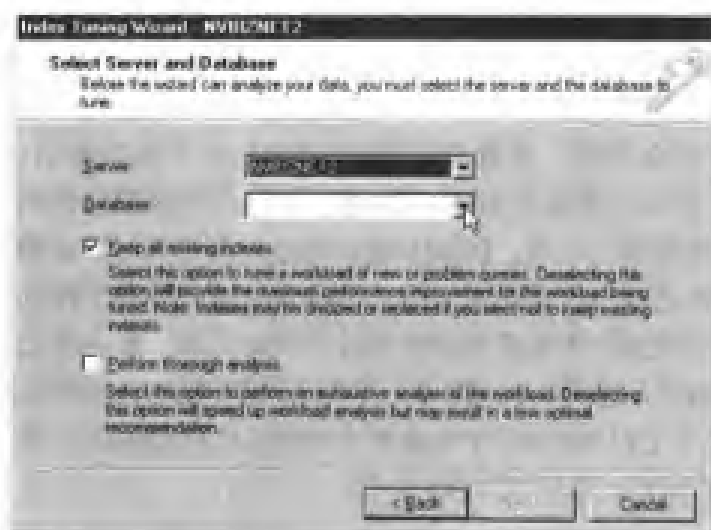


图 7.3 MS-SQL Server 的 Index Tuning Wizard Select Server and Database 对话框

8. 为了选择包括想要创建索引的表的数据库，可单击 Database 域右边的下拉按钮。在 Index Tuning Wizard 显示在步骤 3 中所选的服务器上的数据库名清单上，单击想要使用的数据库名。对本例来说，单击 SQLTips。

9. 为了让 Index Tuning Wizard 删除它所发现的基于系统查询负载来说不必要的已有索引，可单击 Keep all existing indexes 左边使选择标记消失。

注意：Index Tuning Wizard 可能会删除 SELECT 语句的 INDEX=<index name>子句中的索引名。结果，以前功能正常的查询可能会不正常。由于 MS-SQL Server Query Optimizer（查询优化器）几乎总是为查询选择最有效的索引，所以最好不要让 SQL Server 强制使用特定的索引（通过在 SELECT 语句中包括 INDEX=<index name>子句）。

10. 为了让 Index Tuning Wizard 提出会改善系统和查询性能的索引的建议，可单击 Perform thorough analysis（执行全面分析）左边的复选框使之出现选择标记。

11. 单击 Next 按钮。该向导将显示 Index Tuning Wizard—Identify Workload（索引调节向导—确定工作负载）对话框，如图 7.4 所示。

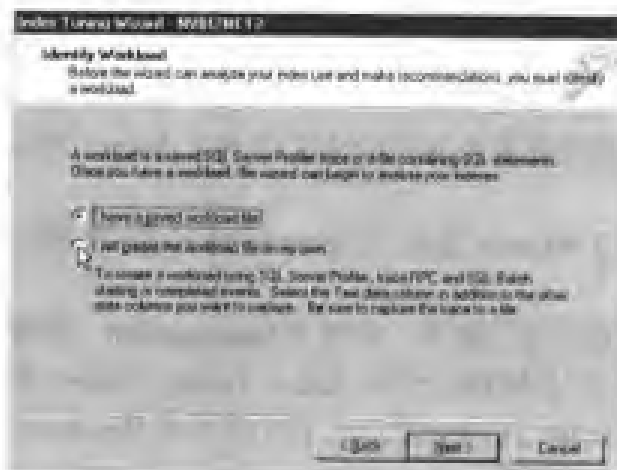


图 7.4 MS-SQL Server Index Tuning Wizard 的 Identify Workload 对话框

12. 如果以前在磁盘上保存过 MS-SQL Server Profiler 跟踪文件或是有一个包括有让 Index Tuning Wizard 据以分析的 SQL 语句, 则单击 I Have a Saved Workload File (我有保存的工作负载文件) 单选按钮, 然后单击 Next 按钮并从步骤 21 处继续。

13. 为了创建一个系统负载的新跟踪文件, 可单击 I Will Create the Workload File on My Own (我将创建自己的工作负载文件) 单选按钮, 然后单击 Finish 按钮。Index Tuning Wizard 将结束, 而 Enterprise Manager 将启动 MS-SQL Server Profiler。

14. 在 MS-SQL Server 的 Profiler 应用程序窗口, 选择 File 菜单中的 Run Traces (运行跟踪) 选项 (或在标准工具栏上单击 Start Traces [绿色的三角形] 按钮)。MS-SQL Server Profiler 将在 Start Selected Traces (启动所选的跟踪) 对话框 (如图 7.5 所示) 中显示跟踪文件的清单。



图 7.5 MS-SQL Server Profiler 的 Start Selected Traces 对话框

15. Sample 1 跟踪文件 (是 MS-SQL Server 所带的标准文件) 将捕获事件和 Index Tuning Wizard 分析所需的数据列。由此, 可单击 Sample 1-TSQL (<database name>) 来选择 Sample 1 跟踪文件, 然后单击 OK 按钮。

MS-SQL Server Profiler 将对步骤 8 中所选的数据库启动由 SQL 服务器所处理的 SQL 查询的记录。必须让 Profiler 运行一会才能在 DBMS 上建立起拥有数据库查询工作负载的代表性示例的日志。根据系统使用率情况, 捕获代表性示例工作负载可能要花几小时或几天。在正常操作期间捕获经常使用的好的查询代表例是非常重要的, 因为 Index Tuning Wizard 是根据日志文件中优化的 SELECT 语句提出其建议的。如果日志不是常常出现在 DBMS 中的查询的代表, 那么 Index Tuning Wizard 所建议的索引在系统正常工作负载下可能不会产生好的性能。

注意: 如果正在长时期运行 Server Profiler, 一定要通过选择 File 菜单中的 Save 选项 (至少) 每几小时将跟踪日志内容周期性地写入硬盘。当出现 Save As 对话框时, 在 File Name 域内键入跟踪文件名, 然后单击 OK 按钮。要避免的事情是创建大型的未保存的日志文件, 仅让系统由于某种原因在将跟踪文件保存到磁盘之前重置或死锁。

16. 在跟踪文件包括了有代表性的系统工作负载示例之后, 单击标准工具栏上的 Stop This Trace (停止本次跟踪) (红色的方块) 按钮。

17. 为了将跟踪日志保存到磁盘上, 请选择 File 菜单中的 Save 选项。MS-SQL Server Profiler 将显示 Save As 对话框。

18. 在 File Name 域内键入日志文件名，然后单击 OK 按钮。对本例来说，在 File Name 域内键入 INDEX_TRACE-SQLTips。

19. 选择 File 菜单中的 Exit 选项，退出 MS-SQL Server Profiler 应用程序。

20. 返回 MS-SQL Server Enterprise Manager 窗口并重复步骤 4~步骤 12（如果在运行 Profiler 时关闭了 Enterprise Manager，可重复步骤 1~步骤 12。在执行步骤 12 时，一定要单击 I Have a Saved Workload File 单选钮。

21. 在单击了步骤 12 中的 Next 按钮之后，Index Tuning Wizard 将显示 Specify Workload（指定工作负载）对话框，如图 7.6 所示。单击 My Workload File（我的工作负载）单选钮。向导将显示 Open 对话框。

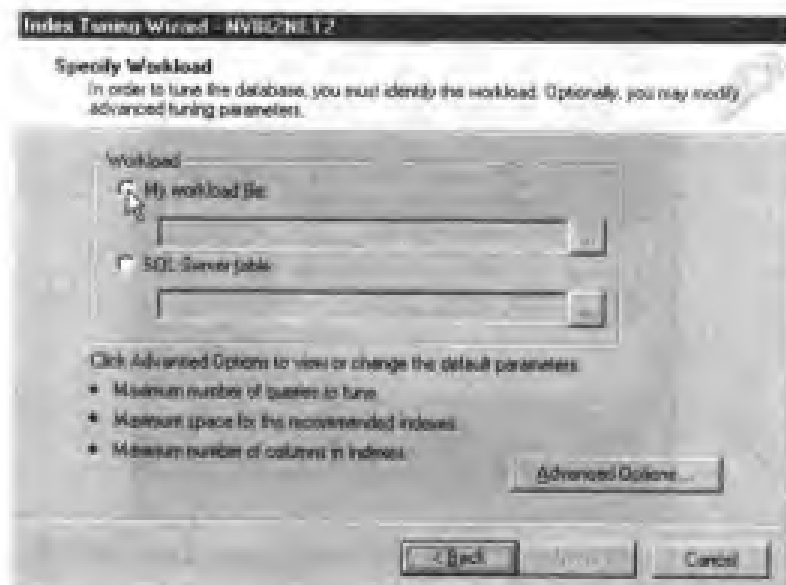


图 7.6 MS-SQL Server Index Tuning Wizard 的 Specify Workload 对话框

22. 在 File Name 域中键入想要 Index Tuning Wizard 分析的跟踪文件，然后单击 OK 按钮。对本例来说，在 File Name 域中键入 INDEX_TRACE-SQLTips。

23. 如果想要指定要调节的最大查询数、用于所建议索引的最大空间或是每索引的最大列数，可单击 Advanced Options（高级选项）按钮。Index Tuning Wizard 将显示其 Index Tuning Parameters 对话框，其中有必须键入以上 3 个最大值的每一个的选项。对本例来说，可接受系统默认值。

注意：在这 3 种最大选项中，如果数据库表有大量的行，最好增加用于建议索引的最大空间——这就意味着每个索引需要保存大量的键值。要调节的查询的最大数目和每索引的最大列数的默认值通常可产生最佳的调节结果。如果不能确定想要的设置，总可以通过重复步骤 23~步骤 25 尝试不同的设置，并使用靠近 Index Recommendations（索引建议）对话框（如图 7.8 所示）底部显示的估计改善百分比的最高值时的设置。

24. 单击 Next 按钮显示 Index Tuning Wizard-Select Tables to Tune（索引调节向导-选择要调节的表）对话框，如图 7.7 所示。

默认值是让向导调节所有的表。如果只想处理某些表，可在对话框的 Tables to Tune（要调节的表）部分中选择那些不想要调节的表，然后单击 Remove 按钮，排除从索引过程中选择的表。

对本例来说，可接受默认值并让 Index Tuning Wizard 调节 SQLTips 数据库中的所有表的索引。

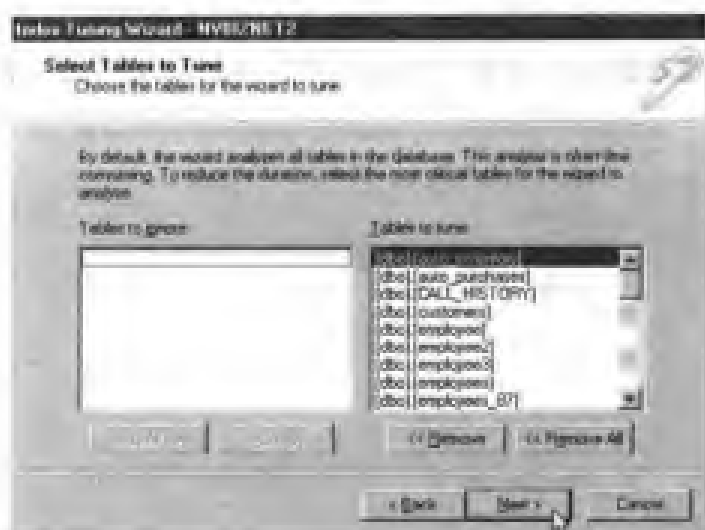


图 7.7 MS-SQL Server Index Tuning Wizard 的 Select Tables to Tune 对话框

25. 在对话框的 Tables to Tune 部分列出了想要调节索引的表之后，单击 Next 按钮。Index Tuning Wizard 将读取在步骤 22 中键入的日志文件并使用 MS-SQL Server 的 Query Optimizer 根据使用在 SELECT 语句中列的每种可能的索引组合来估计每条查询的可能的性能。在向导完成了其分析之后，将显示日志文件中整个查询集产生最佳整体性能的索引清单，如图 7.8 所示。

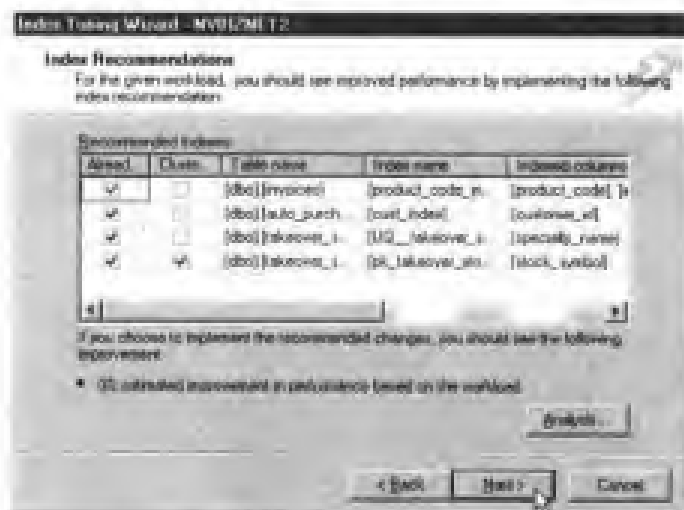


图 7.8 MS-SQL Server Index Tuning Wizard 的 Index Recommendations 对话框

26. 单击 Next 按钮。向导将显示 Index Tuning Wizard-Schedule Index Update Job（索引调节向导-安排索引更新任务）对话框，如图 7.9 所示。

27. 为了让 Index Tuning Wizard 将建议的改变应用于数据库的索引结构，可单击 Apply changes 复选框使之出现选择标记。接着，决定是否现在就要索引过程发生还是想要将其安排在其他时间。对本例来说，单击 Execute Recommendations Now（现在执行建议）左边的单选框。

注意：在许多（或大型）数据库表上创建索引将对系统性能有实质性影响，因为 DBMS 将要把主要的处理时间花在此任务上，用户将发现其语句执行得比通常要慢。另外，当 DBMS 对已有的表创建集群索引时，表在创建索引时是不可使用的。因而，要仔细估计系统的使用情

况并在系统使用率低或不使用（最好）的情况下创建索引，这样对数据库用户和应用程序的影响最小。

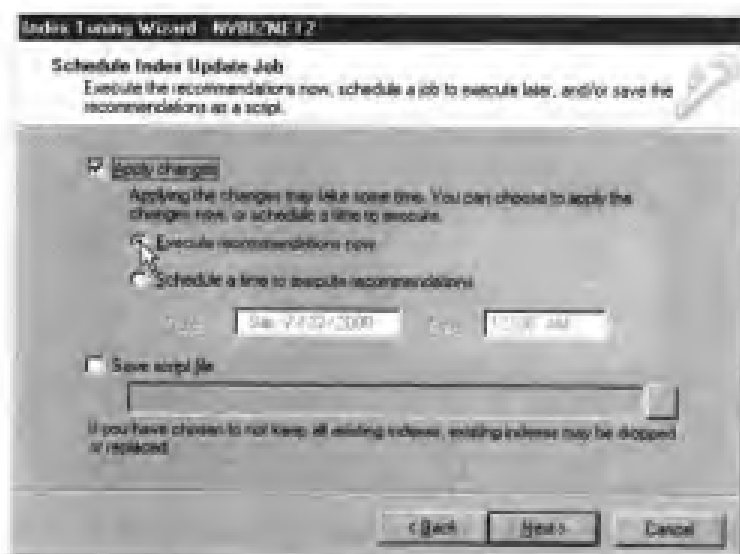


图 7.9 MS-SQL Server Index Tuning Wizard 的 Schedule Index Update Job 对话框

28. 在选择了索引创建时间之后，单击 Next 按钮。Index Tuning Wizard 将显示 Completing the Index Tuning Wizard（完成索引调节向导）消息框。单击 Finish 按钮，或是当时就创建索引或是在以后创建，这要由步骤 27 中的选择来决定。

在步骤 28 之后，Index Tuning Wizard 显示其成功地完成消息框时，单击 OK 按钮返回 Enterprise Manager 应用程序窗口。

第 8 章 使用键字和约束保持数据库的一致性

技巧 127 理解单列和复合键字

键字是其值可惟一地确定表中的一行的列或列组合。如果知道键值并将其用作 SELECT 语句中以下形式的相等准则：

```
SELECT * FROM <table name> WHERE <key column> = <key value>
```

则 DBMS 将显示表中单行的内容。如果 DBMS 返回不止一行，则所选的列不是 SELECT 语句的 FROM 子句中指定的表的单列键字。

例如，假设有用以下语句创建的 EMPLOYEE 表：

```
CREATE TABLE employee
  (emp_num SMALLINT,
   name VARCHAR(30),
   address VARCHAR(50),
   payrate MONEY)
```

虽然 EMPLOYEE 表中的任一列理论上都可用作键字，但是，例如使用 NAME 就要求公司中只有一个人名为 John Smith。如果公司雇用了第二个 John Smith，DBMS 就不能再使用基于 NAME 列的单列键字，因为 DBMS 不能区分 EMPLOYEE 表中的第一个 John Smith 行和第二个 John Smith 行：

emp_num	name	address	payrate
1	Konrad King	765 E. Eldorado Lane	10.25
2	Karen King	765 E. Eldorado Lane	25.75
3	John Smith	2234 State Street	12.35
4	Harry Smith	2258 Hilly Way	10.25
5	John Smith	589 Hillcrest Drive	32.58

类似地，ADDRESS 或 PAYRATE 也都不是好的键字候选列，因为两者都在表的不同行上有重复的值。如果使用 ADDRESS 列作为键字，DBMS 不能区分表中的第一行和第二行。这样一来，EMP_NUM 是 EMPLOYEE 表中惟一适合于单列键字的列，因为 EMP_NUM 在表的每一行上有惟一的值。

不管能否将 EMP_NUM 列确定为 PRIMARY_KEY，以下 SELECT 语句将显示 EMPLOYEE 表中单行的内容：

```
SELECT * FROM EMPLOYEE WHERE emp_num = 3
```

向那些其值在表的每一行上都是惟一的列施加 PRIMARY KEY 约束让数据库的用户知道哪一列可用来找出表中的特定的行，而且还可防止任何人用该列的 NULL 值或是重复的值来添加行。

有时表中没有单列在每一行中都有惟一的值。例如，假设公司在不止一个地方有办公室，每个都有其自己的惟一一套雇员号码，必须使用多列或复合键字。

在本例中，EMPLOYEE 表的复合键字可以是 EMP_NUM 和 OFFICE。如果知道复合键字中的列值，就可在 SELECT 语句中以以下形式将其用作相等标准：

```
SELECT * FROM <table name>
WHERE <first key column> = <first column key value> AND
      <second key column> = <second column key value>
```

以便显示出表中的特定的一行（如果复合键字不止两列，应将另外的 AND 关键词添加到相等标准中复合键字中的第 3 个和以后的列）。

正如单列键字的情况一样，以下 SELECT 语句将显示 EMPLOYEE 表中单列的内容，而不管是否向该列施加了 PRIMARY KEY 约束：

```
SELECT * FROM EMPLOYEE WHERE emp_num = 3 AND office = 'TX'
```

但是，如果施加了键字约束，数据库用户将知道使用哪个列组合才能找出表中的特定的行。另外，PRIMARY KEY 约束将防止某人在 EMP_NUM 或 OFFICE 列中用 NULL 值来添加新行以及以与已有雇员相同的 EMP_NUM 或 OFFICE 列值添加新雇员。

技巧 128 使用 CREATE DOMAIN 语句创建域

域是列可保存的合法数据值的集合。当把列定义为标准 SQL 或是具体的 SQL 服务器特有的数据类型（INTEGER、NUMERIC、CHARACTER、DATETIME 等）时，就要指定该列的合法值的集合（域）。例如，MS-SQL Server 有 3 种 INTEGER 数据类型：INTEGER（或 INT）、SMALLINT 和 TINYINT。当定义类型为 INTEGER 的列时，可将其域设置为范围为 -2,147,483,468~2,147,483,647 内的整数。类型为 SMALLINT 的列的域为范围为 -32,768~32,767 中的整数。同时，TINYINT 列的域为范围为 0~255 中的整数。

但是，有时需要将列域限制到特定数据类型的全部合法值的子集，这正是 CREATE DOMAIN 语句的目的。SQL-92 的 CREATE DOMAIN 语句允许对标准的 DBMS 数据类型应用 CHECK 约束创建一个新数据类型，从而限制其域。

CREATE DOMAIN 语句的句法如下：

```
CREATE DOMAIN <domain name> AS <data type>
[DEFAULT <default value>]
[<first constraint definition>
...<last constraint definition>]
```

其中：

```
<constraint definition> ::=
[CONSTRAINT <constraint name>]
CHECK (VALUE <conditional expression>)
```

例如，假设公司的雇员号是以 1 开头的 4 位数。以下 CREATE 语句允许插入非法的雇员号——那些超出了范围 1000~1999 的号码：

```
CREATE TABLE employees
(employee_num INTEGER,
name CHAR(30),
address CHAR(50))
```

为了将 EMPLOYEE_NUM 列的域限制为范围 1000~1999，可使用以下语句创建域：

```
CREATE DOMAIN valid_emprnums
```

```
CHECK (VALUE BETWEEN 1000 AND 1999)
```

然后使用域代替表定义中的 INTEGER 类型:

```
CREATE TABLE employees
  (employee_num valid_empnums,
   name CHAR(30),
   address CHAR(50))
```

注意: 并不是所有的 DBMS 产品都支持 CREATE DOMAIN 语句。如果具体的 DBMS 产品不支持, 请检查系统手册上的有关 CHECK 约束以及用户定义的数据类型方面的内容。许多时候可通过创建新数据类型并将 CHECK 约束应用其上, 就可创建命名的域。例如, MS-SQL Server 就没有 CREATE DOMAIN 语句。但是, 在 MS-SQL Server 上仍然可以限制本例中的 EMPLOYEE_NUM 的域, 只要执行以下 Transact-SQL 语句:

```
EXEC sp_addtype valid_empnums, 'SMALLINT'
CREATE RULE validate_empnum
AS @employee_number BETWEEN 1000 AND 1999
EXEC sp_bindrule 'validate_empnum', 'valid_empnums'
```

执行以上 3 条语句的总结果与执行 CREATE DOMAIN 语句是相同的——结果都是有了 VALID_EMPNUMS 数据类型, 其域是范围为 1000~1999 中的整数。

技巧 129 使用 PRIMARY KEY 列约束惟一地确定表行

如果表有 PRIMARY KEY 或者至少一列被约束为 UNIQUE 和 NOT NULL, 则表中的每行是惟一的。因此, 通过指定行的 PRIMARY KEY 值作为 SQL 语句的 WHERE 子句中的相等条件从而更新列、显示或删除一个表中特定的行都是可能的。

例如, 假设在由以下语句创建的 EMPLOYEES 表中有一个单列 PRIMARY KEY:

```
CREATE TABLE employees
  (emp_num SMALLINT,
   office CHAR(2),
   name VARCHAR(50),
   address VARCHAR(50),
   payrate MONEY,
   PRIMARY KEY (emp_num))
```

对 EMP_NUM 列的 PRIMARY KEY 约束意味着, 表中的每一行都有不同的非 NULL 的 EMP_NUM 列值。由此, 执行以下 SELECT 语句将显示 0 或 1 行:

```
SELECT * FROM employees WHERE emp_num = 1001
```

如果没有雇员 1001, 那么 DBMS 将显示 0 行, 如果有雇员 1001, DBMS 将显示单行, 因为 1001 在表的 EMP_NUM 列中只能出现在一行上。

类似地, 如果有两个列 PRIMARY KEY, 如下面的定义那样:

```
CREATE TABLE employees
  (emp_num SMALLINT,
   office CHAR(2),
   name VARCHAR(30),
   address VARCHAR(50),
   payrate MONEY,
   PRIMARY KEY (emp_num, office))
```

一条 SQL 语句通过执行对 PRIMARY KEY 中的每一列中相等测试并将两个等式用布尔运算符 AND 连接起来，即可命中表中的特定行。例如，以下 UPDATE 语句将给出 0 个或一个雇员有 20% 的支付率增加：

```
UPDATE employees SET payrate = payrate * 1.2
WHERE emp_num = 2 AND office = 'LV'
```

对两列 EMP_NUM 和 OFFICE 的 PRIMARY KEY 约束保证至多有一个 EMP_NUM/OFFICE 对。可能有不只一行的 EMP_NUM 列值为 2，而且在 OFFICE 列也有不止一行的值为 LV。但是，对复合（多列）的 PRIMARY KEY 约束——在本例中是两列——却保证了在 EMPLOYEES 表中没有多于一行的 EMP_NUM 列值为 2，而且 OFFICE 列值为 LV。

作为第 3 个例子，假如有 4 列的 PRIMARY KEY，如以下语句所定义的：

```
CREATE TABLE call_history
(phone_number CHAR(7),
called_by CHAR(3),
date_called DATETIME,
call_time SMALLINT,
hangup_time SMALLINT,
disposition CHAR(4),
PRIMARY KEY (phone_number, called_by, date_called,
hangup_time))
```

执行 4 个由 AND 布尔运算符连接在一起的相等测试的一条 SQL 语句——PRIMARY KEY 中的每一列有一个测试——将惟一地命中 CALL_HISTORY 表中的单行。例如，以下 DELETE 语句从 CALL_HISTORY 表中要么删除 0 行要么删除 1 行：

```
DELETE FROM call_history
WHERE phone_number = '3610141' AND called_by = 'KLK' AND
date_called = '07/24/2000' AND hangup_time = 2100
```

与单列和两列的 PRIMARY KEY 约束的情况一样，CALL_HISTORY 表中的 PRIMARY KEY 约束保证，至多有一行有特定的 PHONE_NUMBER 和 (AND) CALLED_BY 和 (AND) DATE_CALLED 和 (AND) HANGUP_TIME 组合。

在本技巧中使用的特定的值、列和表并不是重要的。要记住的是，通过在 WHERE 子句中对 PRIMARY KEY 中的每一列进行的相等测试用 AND 布尔运算符连接起来，可使用 PRIMARY KEY 显示或处理表中一个特定的行。

技巧 130 理解引用完整性检查和外键

FOREIGN KEY 允许建立两个表间的父子关系。不用使用指针把一个表中的父行与另一表中的一行或多行（子）链接起来，关系 DBMS 将父行的 PRIMARY KEY 值存储在子行的 FOREIGN KEY 列中。关系数据库准则规定子表中的每个 FOREIGN KEY 值必须作为 PRIMARY KEY 值存在于其父表中，这条准则叫做引用完整性约束。

例如，如果有两个用以下语句创建的 EMPLOYEES 和 TIMECARDS 表：

```
CREATE TABLE employees
(employee_number SMALLINT,
name VARCHAR(30),
payrate MONEY,
```



```
PRIMARY KEY (employee_number))
CREATE TABLE timecard
(emp_no SMALLINT,
card_date DATETIME,
time_in SMALLINT,
time_out SMALLINT,
CONSTRAINT fk_employees_timecard FOREIGN KEY (emp_no)
REFERENCES employees (employee_number))
```

对 TIMECARD 表中的 EMP_NO 列的 FOREIGN KEY 约束告诉 DBMS 确保 TIMECARD 表中的每一行中的 EMP_NO 列值有与 EMPLOYEES 表中的一行的 EMPLOYEE_NUMBER 列 (PRIMARY KEY 列) 相匹配的值。

保持一个表中的 FOREIGN KEY 列与另一表中的 PRIMARY KEY 列的引用完整性涉及对以下 4 类有可能破坏父/子关系的表更新中的每一类执行引用完整性检查:

- 向子表中插入行。DBMS 必须确保插入到子表中的每一新行具有与父表中的 PRIMARY KEY 值匹配的 FOREIGN KEY。如果新行中的 FOREIGN KEY 值与父表中的 PRIMARY KEY 值之一并不匹配, DBMS 中止 INSERT 语句。向子表中插入不匹配的 FOREIGN KEY 值可能会破坏引用完整性, 因为创建了“孤行”, 即没有父记录的子记录, 这是不允许的。
- 从父表中删除一行。DBMS 将中止任何企图从父表中删除行, 如果要被删除行的 PRIMARY KEY 值作为 FOREIGN KEY 出现在子表的行中。从父表中删除与 PRIMARY KEY 匹配的行会由于创建了“孤行”(即子记录中的 FOREIGN KEY 值没有与之匹配的父表中的 PRIMARY KEY 值)而破坏引用完整性, 因而也是不允许的(在技巧 136 “理解如何应用 CASCADE 规则更新与删除以帮助保持引用完整性”中, 读者将学习某些 DBMS 产品如何对 DELETE 语句应用 CASCADE 规则来删除父行及其所有的子行, 从而允许 DELETE 语句来删除在其他表中有子行的父表行)。
- 更新子表中的 FOREIGN KEY 值。不管是新行还是已有的行, 行中的 FOREIGN KEY 值必须与由 FOREIGN KEY 约束引用的父表中的 PRIMARY KEY 值相匹配。由此, 如果 UPDATE 语句试图将子行中的 FOREIGN KEY 值改变为与父表中的 PRIMARY KEY 值不再匹配, 则 DBMS 中止该 UPDATE 语句。
- 更新父表中的 PRIMARY KEY 值。改变父表中的 PRIMARY KEY 值与删除有原来的 PRIMARY KEY 值和插入有新的 PRIMARY KEY 的另一行有相同的效果。因此, 如果当前 PRIMARY KEY 值作为 FOREIGN KEY 值出现在子表行中, 则 DBMS 中止企图改变一行的 PRIMARY KEY 值的 UPDATE 语句。将在父表中与 PRIMARY KEY 匹配的值改为新值会破坏引用完整性, 因为创建了“孤行”, 即子记录(行)中的 FOREIGN KEY 值在父表的 PRIMARY KEY 列中没有与之匹配的值, 因而这是不允许的(在技巧 136 中, 读者将了解, 某些 DBMS 产品将向 UPDATE 语句应用 CASCADE 规则并改变父行中的 PRIMARY KEY 值以及子行中的每一个 FOREIGN KEY 值, 从而允许 UPDATE 语句改变父表行中的 PRIMARY KEY 值和其他表中的子行)。

有关引用完整性的重要事项是, 每当 UPDATE 语句企图改变表的 PRIMARY KEY 中的一列或多列, 或者每当 DELETE 语句企图从有 PRIMARY KEY 约束的表中删除一行时, DBMS

检查其系统表，看是否有引用了 PRIMARY KEY 的 FOREIGN KEY。作为对照，DBMS 在执行 UPDATE 语句来改变 FOREIGN KEY 中的一列或多列或者执行 INSERT 语句向有 FOREIGN KEY 约束的表中插入新行之前，都要检查 PRIMARY KEY 索引，确保 FOREIGN KEY 列中的值存在于 PRIMARY KEY 索引中。

如果执行 UPDATE、INSERT 或 DELETE 语句会制造“孤行”，即没有相应的 PRIMARY KEY 值（在父表中）的 FOREIGN KEY 值（在子表中），DBMS 将中止语句的执行，以便保持引用数据的完整性。

技巧 131 理解引用数据完整性检查为什么会危害安全性

在技巧 110 “使用 GRANT SELECT（以及 REVOKE SELECT）语句控制对数据库对象的访问”中，读者已经学习了如何通过向某些用户 ID 授予对表的访问权限让他们看到表中的信息。还学习了如何将表的内容对其他用户 ID 隐藏起来，即不向那些用户授予对表的 SELECT 访问权，或者撤消以前授予的 SELECT 访问权。遗憾的是，对表没有 SELECT 访问权的用户 ID 可使用引用数据完整性检查（在技巧 130 中所学习的）来推出表的 PRIMARY KEY 列的值。

例如，假设读者是娱乐场所负责游戏者关系的负责人，而且要将顶级游戏者保存到以下面语句创建的表中：

```
CREATE TABLE high_rollers
  (player_ID INTEGER, name VARCHAR(30),
   credit_limit MONEY,
   average_action MONEY,
   YTD_winnings MONEY,
   YTD_losses MONEY,
   PRIMARY KEY (player_ID))
```

如果饭店在用以下语句创建的表中跟踪所有游戏者：

```
CREATE TABLE big_gamblers
  (player_number INTEGER,
   name          VARCHAR(30),
   address       VARCHAR(50),
   phone_number  VARCHAR(20),
   PRIMARY KEY (player_number))
```

对 HIGH_ROLLERS 表只有 REFERENCES 权限而对 PLAYER_CARDS 表有 SELECT 访问权的一位雇员，通过使用以下语句创建一个引用 HIGH_ROLLERS 表中的 PRIMARY KEY 的表：

```
CREATE TABLE big_gamblers
  (player_number INTEGER,
   name VARCHAR(30),
   address VARCHAR(50),
   phone_number VARCHAR(20),
   CONSTRAINT fk_high_roller_id FOREIGN KEY (player_number)
   REFERENCES high_rollers (player_ID))
```

通过以下语句列出每个 CUSTOMER_ID：

```
SELECT customer_ID FROM players_cards
```

然后对列出的每个 CUSTOMER_ID 试用以下 INSERT 语句：

```
INSERT INTO big_gamblers
VALUES (<customer_ID from players_cards table>, NULL, NULL,
      NULL)
```

就可以列出顶级游戏者的清单。

因为 FOREIGN KEY 约束将允许 DBMS 仅当要插入的行中的 PLAYER_NUMBER 列值存在于 HIGH_ROLLERS 表中一行的 PLAYER_ID 列时,才可向 BIG_GAMBLERS 表插入行,所以 BIG_PLAYERS 表将包括来自 PLAYERS_CARDS 表中的所有 CUSTOMER_ID 的清单,而这同时也是 HIGH_ROLLERS 表中的 PLAYER_ID。

向 BIG_GAMBLERS 表填入其余信息可由以下 INSERT 语句完成:

```
INSERT INTO big_gamblers
SELECT player_cards.customer_ID, player_cards.name,
player_cards.address, player_cards.phone_number
FROM big_gamblers, player_cards
WHERE player_cards.customer_ID = big_gamblers.player_ID
```

然后使用以下 DELETE 语句只删除带有那些顾客 ID 的行:

```
DELETE FROM big_gamblers WHERE name IS NULL
```

从本例中要理解的重要事情是,一个用户可使用 REFERENCES 权限和对 INSERT 语句的引用完整性检查来推出表中的 PRIMARY KEY 列的值(在 MS-SQL Server 上也可推出带有 UNIQUE 约束的任何列的值)。因而,一定要保护表数据,只向那些允许显示 PRIMARY KEY 和 UNIQUE 约束列值的用户授予 REFERENCES 权限,从而不会在无意中泄露数据。

技巧 132 理解引用完整性检查如何限制删除行和表的能力

正如在技巧 130 “理解引用完整性检查和外键”中所学习的,引用完整性检查是 SQL 服务器采取的用来保持表间的(父/子)关系的行动。既可由单列也可由多列值(多列的复合值)组成的 FOREIGN KEY 要与另一表中的 PRIMARY KEY 值相匹配。换句话说,通过向一个表中添加一列或多列并在其中存储所链接的另一表中的行的 PRIMARY KEY 值从而建立两表之间的链接。在对所添加的列应用了 FOREIGN KEY 约束之后,DBMS 确保列中的该值总是与另一表中的 PRIMARY KEY 列值相匹配。

注意:在 MS-SQL Server 上,可对有 UNIQUE 约束的表列以及带有 PRIMARY KEY 约束的表列建立 FOREIGN KEY 引用。

为了保持引用完整性,DBMS 检查每条涉及有一条 PRIMARY KEY 或另一 FOREIGN KEY 约束的表的 INSERT、UPDATE、DELETE 和 DROP 语句。DBMS 中止任何引起存储引用了不存在的 PRIMARY KEY 值的 FOREIGN KEY 值的语句。例如,给定用以下语句创建的表:

```
CREATE TABLE item_master
(
  product_code INTEGER,
  description   VARCHAR(30),
  bin_level     SMALLINT,
  PRIMARY KEY (product_code)
)
CREATE TABLE inventory
(
  item_number   INTEGER,
  supplier_code INTEGER,
  cost          MONEY,
```

```
qty_on_hand    SMALLINT,  
CONSTRAINT fk_item_master_inv FOREIGN KEY (item_number)  
REFERENCES item_master (product_code))
```

DBMS 将中止试图删除其 PRODUCT_CODE (PRIMARY KEY) 值与 INVENTORY 表中的 ITEM_NUMBER (FOREIGN KEY) 值相匹配的 ITEM_MASTER 行的 DELETE 语句。由此，如果 INVENTORY 表有在 ITEM_NUMBER 列值为 1001 的行，DBMS 将中止以下 DELETE 语句：

```
DELETE item_master WHERE PRODUCT_CODE = 1001
```

并显示下面的错误信息：

```
Server: Msg 547, Level 16, State 1, Line 1  
DELETE statement conflicted with COLUMN REFERENCE  
constraint 'fk_item_master_inv'. The conflict occurred in  
database 'SQLTips', tab 'inventory', column  
'item_number'.  
The statement has been terminated.
```

在本例中，为了成功地执行该 DELETE 语句，必须首先删除 INVENTORY 表中所有 ITEM_NUMBER 列值为 1001 的行，或者使用 ALTER TABLE 语句删除 INVENTORY 表的 FK_ITEM_MASTER_INV 列上的 FOREIGN KEY 约束。

类似地，DBMS 将中止任何企图删除被 FOREIGN KEY 约束引用的表——即使该表的 FOREIGN KEY 引用是空的也不行。例如，在本例中给定的表，DBMS 将中止以下 DROP TABLE 语句：

```
DROP TABLE item_master
```

出现错误信息如下：

```
Server: Msg 3726, Level 16, State 1, Line 1  
Could not drop object 'item_master' because it is  
referenced by a FOREIGN KEY constraint.
```

如果想要删除 (DROP) ITEM_MASTER 表，必须首先删除引用了该表的 FOREIGN KEY 约束。遗憾的是，在试图删除被 FOREIGN KEY 约束引用的表时所生成的错误信息不一定是明确地指出包括 FOREIGN KEY 引用的表（因而要想删除被 FOREIGN KEY 引用的表，必须知道有 FOREIGN KEY 引用的表，这样才能执行 ALTER TABLE 语句来删除那些表中的 FOREIGN KEY）。请检查一下数据字典，其中应该说明所有表的依赖性。或者使用系统工具之一让 DBMS 列出表引用。例如，在 MS-SQL Server 上，可用以下语句执行 sp_help 存储过程：

```
EXEC SP_HELP item_master
```

然后查看输出的 Table Is Referenced By(表被……引用)部分，找出引用了 ITEM_MASTER 表的所有数据库对象和 FOREIGN KEY 约束。

技巧 133 理解引用完整性检查的 INSERT 死锁及解决办法

引用完整性检查保证一个表中的 FOREIGN KEY 有与另一表中的 PRIMARY KEY 值相匹配的值。遗憾的是，当一个表中的 FOREIGN KEY 引用了另一表中的 PRIMARY KEY，而第二个表中的 FOREIGN KEY 又引用了第一个表中的 PRIMARY KEY，这时保持引用完整性会引起 INSERT 语句死锁。这种情况如图 8.1 所示。

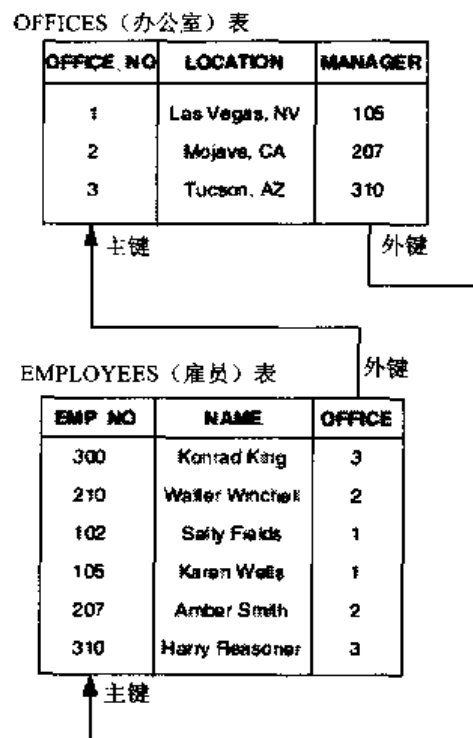


图 8.1 两个表中的循环 FOREIGN KEY 引用

例如，假设公司雇用了一个新雇员去夏威夷的檀香山开办一个新的办公室（4 号办公室）。用以下的 INSERT 语句添加新雇员将不能执行：

```
INSERT INTO employees VALUES (400, 'Debbie Jamsa', 4)
```

因为在 OFFICES 表中没有 4 号办公室的 OFFICE_NO 列值。

因而 INSERT 语句不能进行引用完整性检查。但是，如果试图使用以下 INSERT 语句添加有关 4 号办公室的信息时：

```
INSERT INTO offices VALUES (4, 'Honolulu, HI', 400)
```

DBMS 中止 INSERT 语句，因为违反了对 MANAGER 列的引用完整性检查，因为在 EMPLOYEES 表中没有 EMP_NUM 列值为 400。

这样一来，看起来好像没有办法向数据库中添加新办公室或经理了！

为了防止插入死锁，应确保在两个表中至少有一个受 FOREIGN KEY 约束而互相引用的列允许 NULL 值。

因而，如果在本例中的表是由以下语句创建的：

```
CREATE TABLE offices
  (office_no SMALLINT,
   location VARCHAR(30),
   manager SMALLINT,
  PRIMARY KEY (office_no))
CREATE TABLE employees
  (emp_no SMALLINT,
   name VARCHAR(30),
   office SMALLINT NOT NULL,
  PRIMARY KEY (emp_no),
```

```
CONSTRAINT fk_offices_emp  
    FOREIGN KEY (office) REFERENCES offices (office_no))  
ALTER TABLE offices ADD CONSTRAINT fk_employees_ofce  
    FOREIGN KEY (manager) REFERENCES employees (emp_no)
```

起初仍然不能插入（INSERT）雇员信息，但执行以下语句序列：

```
INSERT INTO offices VALUES (4, 'Honolulu, HI', NULL)  
INSERT INTO employees VALUES (400, 'Debbie Jamsa', 4)  
UPDATE offices SET manager = 400 WHERE office = 4
```

就可创建一个新的办公室，添加新雇员记录，然后将新雇员指定为办公室经理。

注意：与 PRIMARY KEY 列不同，DBMS 允许 FOREIGN KEY 列包括 NULL 值。起初看起来似乎是违反引用完整性。毕竟，如果 FOREIGN KEY 为 NULL，而没有 PRIMARY KEY 列可为 NULL 值，那么，NULL 的 FOREIGN KEY 永远不能与对应的 PRIMARY KEY 值相匹配。对吗？对此问题的回答依赖于 NULL 的“真正”值。请记住，NULL 意味着缺少或未知。由此，DBMS 使得 FOREIGN KEY 列中的 NULL 值有了受到质疑的好处。假定一旦解决了 NULL 值，非 NULL 的 FOREIGN KEY 将成为 PRIMARY KEY 列中的值之一。

如果在循环引用中涉及的表中所有的 FOREIGN KEY 列都受 NOT NULL 约束，当 DBMS 产品允许临时挂起引用完整性检查时，用户仍然可避免 INSERT 死锁。例如在 MS-SQL Server 上，可执行以下语句序列：

```
ALTER TABLE employees NOCHECK CONSTRAINT fk_offices_emp  
INSERT INTO employees VALUES (400, 'Debbie Jamsa', 4)  
INSERT INTO offices VALUES (4, 'Honolulu, HI', NULL)  
ALTER TABLE employees CHECK CONSTRAINT fk_offices_emp
```

第一条 ALTER TABLE 语句挂起了对表 OFFICES 中的 FOREIGN KEY 约束（FK_OFFICES_EMP）的引用完整性检查。这就允许用还没有存在于 OFFICES 表的 PRIMARY KEY（OFFICE_NO）列的值的 OFFICE 列值来插入雇员。在完成插入所要插入的行之后，一定要重新激活引用完整性检查，如在本例中的最后的 ALTER TABLE 语句中所做的那样。

技巧 134 理解 NULL 值与惟一性的相互作用

允许 NULL 值出现在 PRIMARY KEY 列或出现在具有 UNIQUE 约束的列上，会对 DBMS 上的完整性检查机制造成问题。如果表中的一行在其 PRIMARY KEY 列有 NULL 值，那么 PRIMARY KEY 列的值在表的每一行还真正是惟一的吗？答案依赖于缺少的（NULL）数据的“真正”值。

读者在技巧 133“理解引用完整性检查的 INSERT 死锁及解决办法”中不是学过吗？DBMS 将允许 FOREIGN KEY 列有 NULL 值并满足引用完整性要求，只要假设缺少的（NULL）数据的“真正”值与被 FOREIGN KEY 引用的 PRIMARY KEY 中的值之一相匹配。将这同一原因用于 PRIMARY KEY 中的 NULL 值就意味着 NULL 将是键字中的其他值之一的重复值，因而破坏了表的实体完整性。

没有支持主键的 DBMS 产品允许在 PRIMARY KEY 约束中指定的任何列存储 NULL 值。

但是，某些 DBMS 确实允许向有 UNIQUE 约束中插入单个 NULL 值，或者向定义为 UNIQUE 的索引中插入单个 NULL 值。例如，MS-SQL Server 在将 NULL 添加到 UNIQUE 列或索引时，将 NULL 看作为惟一值。由此，如果用以下语句创建一个表：

```
CREATE TABLE test_null  
(row_id INTEGER UNIQUE,  
 row_name VARCHAR(30))
```

MS-SQL Server 将允许存储在 TEST_NULL 表的 ROW_ID 列带有 NULL 值的单行。如果试图向 ROW_ID 插入第二个 NULL 值, INSERT 语句将不能执行, 并出现以下错误消息:

```
Server: Msg 2627, Level 14, State 2, Line1  
Violation of UNIQUE KEY constraint  
'UQ_test_null_7720AD13'. Cannot insert duplicate key in  
object 'test_null'.  
The statement has been terminated.
```

技巧 135 理解如何应用 RESTRICT 规则更新和删除以帮助保持引用完整性

正如读者了解的, 保持引用完整性意味着确保在父表中的 PRIMARY KEY 值与子表中的每个 FOREIGN KEY 相匹配。

由于有 PRIMARY KEY 的表中的每一行有惟一的 PRIMARY KEY 值, 删除有 PRIMARY KEY 值且该值与引用 PRIMARY KEY 的 FOREIGN KEY 中的一个或多个值匹配的行, 会引起引用完整性丧失。删除单个匹配的 PRIMARY KEY 值使得子表中的一行或多行引用不存在的父表中的行 (PRIMARY KEY)。类似地, 改变与对应的 FOREIGN KEY 中的一个或多个值相匹配的 PRIMARY KEY 值——而没有改变那些值使之与 PRIMARY KEY 的新值匹配——也会引起数据库丧失引用完整性。

施加于 FOREIGN KEY 约束上的 RESTRICT 规则告诉 DBMS 中止执行任何删除有子行的父行 (有 PRIMARY KEY 的表中的行) 的 DELETE 语句。因而, 在执行 DELETE 语句时, 如果 DELETE 语句试图删除有与对应的 FOREIGN KEY 中的一个或多个值匹配的 PRIMARY KEY 值的行, RESTRICT 规则通过中止语句的执行告诉 DBMS 保留引用完整性 (同时取消当前事务处理中所完成的任何工作)。

另外, RESTRICT 规则还告诉 DBMS 中止执行任何改变有子行的父行 (有 PRIMARY KEY 的表中的行) 的 UPDATE 语句。因而, 在执行 UPDATE 语句时, 如果 UPDATE 语句试图改变有与对应的 FOREIGN KEY 中的一个或多个值匹配的 PRIMARY KEY 值的行, RESTRICT 规则通过中止语句的执行告诉 DBMS 保留引用完整性 (同时取消当前事务处理中所完成的任何工作)。

大多数 DBMS 产品, 在决定处理违反表间的引用完整性的 UPDATE 或 DELETE 语句时, 默认地使用 RESTRICT 规则。事实上, 某些 DBMS 产品, 包括 MS-SQL Server, 不仅默认地使用 RESTRICT 规则, 而且还禁止选择任何其他可能的规则 (CASCADE、SET NULL 和 SET DEFAULT) 之一。

请参考具体的 DBMS 系统手册, 从而找出哪条规则是本系统的默认规则, 以便确定 DBMS 是否支持其他规则。请检查一下手册的 UPDATE_RULE 或 DELETE_RULE 索引条目。还要查找一下创建 FOREIGN KEY 约束的句法。如果 DBMS 允许在 UPDATE 或 DELETE 语句试图违反引用完整性约束时选择系统采用的规则 (或动作), FOREIGN KEY 约束的创建句法中将包括可选的 ON UPDATE 和 ON DELETE 子句, 如下所示:

```
[ ON UPDATE  
 NO ACTION | CASCADE | RESTRICT | SET NULL | SET DEFAULT ]
```

```
[ON DELETE
```

```
NO ACTION|CASCADE|RESTRICT|SET NULL|SET DEFAULT] ]
```

注意：SQL-92 和 MS-SQL Server 将在本技巧中描述的动作称为 NO ACTION 而不是 RESTRICT。MS-SQL Server 的在线说明书文档指明，“NO ACTION means that no action is performed and the Transact-SQL statement to perform changes does not execute.”（NO ACTION 意味着不执行任何动作，而且执行改变的 Transact-SQL 语句不会执行。）但是，除了不执行语句之外，DBMS 还生成一条错误条件并显示错误消息。这样一来，似乎 RESTRICT 比 NO ACTION 更准确地描述了发生了什么。如果具体的 DBMS（如 DB2）两条规则都支持，惟一的差别是应用每条规则的时间。DBMS 在任何其他列约束之前实施 RESTRICT 规则，而在实施其他列约束之后实施 NO ACTION 规则。在几乎所有情况下两条规则的操作都是一样的。

技巧 136 理解如何应用 CASCADE 规则更新和删除以帮助保持引用完整性

当一个表的 FOREIGN KEY 列中的每个非 NULL 键字有与之对应的一个且只有一个 PRIMARY KEY 条目相匹配时，数据库具有引用完整性。

根据定义，PRIMARY KEY 中的每个键值对于表中的每行来说都是惟一的。结果，如果 DELETE 语句删除了有 PRIMARY KEY 值的一行，而该值又与对应的 FOREIGN KEY 中的一个或多个键值相匹配，则数据库丧失了其引用完整性。删除 FOREIGN KEY 的单一匹配的 PRIMARY KEY 值，DBMS 生成了“孤行”——一个引用了父表中不存在的行（PRIMARY KEY）的子行。类似地，如果 UPDATE 语句改变了父表行中的 PRIMARY KEY 值，而没有同时改变子行中匹配的 FOREIGN KEY 值，则引用完整性就被破坏了。

应用于 FOREIGN KEY 约束中的 CASCADE 规则告诉 DBMS 在执行对父表行的 DELETE 语句时，自动地删除所有的子行，然后再删除父行。通过遵循 CASCADE 规则，DBMS 可以保证绝没有任何孤行存在，因为系统自动地与其父表一起删除了子行。

类似地，当执行改变 PRIMARY KEY 值的 UPDATE 语句时，CASCADE 规则告诉 DBMS 同时更新所有的子行中的 FOREIGN KEY 值，以便与新 PRIMARY KEY 值相匹配。这样一来，CASCADE 告诉 DBMS，在执行 UPDATE 语句时，自动地将与原来的 PRIMARY KEY 值匹配的 FOREIGN KEY 改变为新的 PRIMARY KEY 值，从而保持引用完整性。

当处理试图违反表间的引用完整性的 UPDATE 或 DELETE 语句时为确定采取什么行动，许多 DBMS 产品只应用 RESTRICT 规则（读者已在技巧 135“理解如何应用 RESTRICT 规则更新和删除以帮助保持引用完整性”中学习了有关知识）。但是，DB2 和 INFORMIX 两者都支持级联式的 DELETE 语句。

请参考自己的 DBMS 的系统手册，查看一下哪个规则是默认的，以便决定 DBMS 是否支持 CASCADE 规则。请检查手册的 UPDATE_RULE 或 DELETE_RULE 索引条目。还要复习一下创建 FOREIGN KEY 的句法。如果 DBMS 在执行企图违反引用完整性约束的 UPDATE 或 DELETE 语句时，允许选择系统将采取的规则，则 FOREIGN KEY 的创建句法将包括可选的 ON UPDATE 和 ON DELETE 子句，如下所示：

```
[ ON UPDATE
```

```
NO ACTION|CASCADE|RESTRICT|SET NULL|SET DEFAULT]
```

```
[ ON DELETE
```

```
NO ACTION|CASCADE|RESTRICT|SET NULL|SET DEFAULT] ]
```


技巧 137 理解如何应用 SET NULL 规则更新和删除以帮助保持引用完整性

根据定义, 如果 FOREIGN KEY 中的每个键值或为 NULL 或与其对应的 PRIMARY KEY 中的一个且只有一个相匹配, 则数据库有引用完整性。

应用于 FOREIGN KEY 约束的 SET NULL 规则告诉 DBMS, 当 DELETE 语句删除父行时或当 UPDATE 语句改变父行中的 PRIMARY KEY 列值时, 将所有子行中的 FOREIGN KEY 列设置为 NULL。这样一来, SET NULL 规则告诉 DBMS, 当执行删除有子行的行的 DELETE 语句时, 将子行中的每个 FOREIGN KEY 值改变为 NULL, 从而保持引用完整性。

注意: 在技巧 133 “理解引用完整性检查的 INSERT 死锁及解决办法”中, 读者已经知道, FOREIGN KEY 的值为 NULL 并不违反引用完整性。DBMS “认为”, 当用户或应用程序提供 FOREIGN KEY 的真实值时, FOREIGN KEY 的 NULL 值将会变为父表的 PRIMARY KEY 值之一。

虽然大多数 DBMS 产品在决定处理企图违反表间的引用完整性约束的 UPDATE 或 DELETE 语句时, 只遵循 RESTRICT 规则, 但 DB2 确实支持 DELETE 语句上的 SET NULL 规则。

请参考自己的 DBMS 的系统手册, 查看一下哪个规则是默认的, 以决定 DBMS 是否支持 SET NULL 规则。请检查手册的 UPDATE_RULE 或 DELETE_RULE 索引条目。还要复习一下创建 FOREIGN KEY 的句法。如果 DBMS 在执行企图违反引用完整性约束的 UPDATE 或 DELETE 语句时, 允许选择系统将采取的规则, 则 FOREIGN KEY 的创建句法将包括可选的 ON UPDATE 和 ON DELETE 子句, 如下所示:

```
[ [ON UPDATE  
    NO ACTION|CASCADE|RESTRICT|SET NULL|SET DEFAULT]  
[ON DELETE  
    NO ACTION|CASCADE|RESTRICT|SET NULL|SET DEFAULT] ]
```

技巧 138 理解如何应用 SET DEFAULT 规则更新和删除以帮助保持引用完整性

引用完整性意味着, 子行中的每个非 NULL 的 FOREIGN KEY 值在父表的 PRIMARY KEY 中都有与之匹配的单个值。

应用于 FOREIGN KEY 约束的 SET DEFAULT 规则告诉 DBMS, 当 DELETE 语句删除父行时, 或当 UPDATE 语句改变父行中的 PRIMARY KEY 列值时, 将所有子行中的 FOREIGN KEY 列设置为列的默认值(对于复合[多列] PRIMARY/FOREIGN KEY 对, DBMS 将 FOREIGN KEY 中的每一列设置为默认值)。这样一来, SET DEFAULT 规则告诉 DBMS, 当执行影响父行中的 PRIMARY KEY 值的 DELETE 或 UPDATE 语句时, 将子行中的每个 FOREIGN KEY 值改变为该列的默认值, 从而保持引用完整性。

注意: 在创建带有子行的表时, 要由创建表的人来确保 FOREIGN KEY 中的列的默认值将生成保证存在于父表的 PRIMARY KEY 中的关键字值。例如, 假设 SALESPERSON 表的 PRIMARY KEY 的行中的一列 (EMP_ID) 有值为 999, 其意为“本部领导”。然后就可将 CUSTOMERS 表中的相关列 (SALESPERSON_ID) 设置为默认值 999(既可在 CREATE TABLE 也可在 ALTER TABLE 语句中设置)。然后, 无论何时从 SALESPERSONS 表中删除离开公司的销售人员时, DBMS 将自动地将该销售人员的顾客改变为本部账户, 即 SALESPERSON_ID 999。

如果 FOREIGN KEY 列的默认值生成一个与 PRIMARY KEY 中的值并不匹配的键值，则 DELETE 或 UPDATE 语句的行为就好像应用了 RESTRICT 规则一样——语句中止执行而且 DBMS 取消在当前事务处理中完成的任何工作。

虽然这是 SQL-92 规范的一部分，但没有一种商业 DBMS 产品现在支持在企图违反表间引用完整性的 UPDATE 或 DELETE 语句中应用 SET DEFAULT 规则。

请参考自己的 DBMS 的系统手册，查看一下哪个规则是默认的，以便决定 DBMS 是否支持 SET DEFAULT 规则。请检查手册的 UPDATE_RULE 或 DELETE_RULE 索引条目。还要复习一下创建 FOREIGN KEY 的句法。如果 DBMS 在执行企图违反引用完整性约束的 UPDATE 或 DELETE 语句时，允许选择系统将采取的规则，则 FOREIGN KEY 的创建句法将包括可选的 ON UPDATE 和 ON DELETE 子句，如下所示：

```
| [ON UPDATE  
  NO ACTION|CASCADE|RESTRICT|SET NULL|SET DEFAULT]  
[ON DELETE  
  NO ACTION|CASCADE|RESTRICT|SET NULL|SET DEFAULT] ]
```

技巧 139 使用 Enterprise Manager 在已有表间添加 FOREIGN KEY 关系

技巧 44“使用 CREATE TABLE 语句指定外键约束”已向读者展示了在使用 CREATE TABLE 创建新表时如何定义外键约束，而技巧 42“使用 ALTER TABLE 语句改变主键和外键”向读者展示了如何使用 ALTER TABLE 语句向已有表中添加 FOREIGN KEY 约束。除了这两个基于命令的工具之外，MS-SQL Server 还提供图形方法用于通过 MS-SQL Server Enterprise Manager 创建 FOREIGN KEY 约束。

为了使用 MS-SQL Server Enterprise Manager 来创建 FOREIGN KEY 约束，请执行以下步骤：

1. 为了启动 Enterprise Manager，可单击 Start 按钮，将鼠标指针移动到 Start 菜单的 Programs 上并选择 Microsoft SQL Server 7.0，然后单击 Enterprise Manager。
2. 为了显示 SQL 服务器的列表，可单击 SQL Server Group 左边的加号 (+)。
3. 找出管理着想要建立一个或多个 FOREIGN KEY 约束的表的数据库的 SQL 服务器，单击 SQL Server 左边的加号 (+)。Enterprise Manager 将显示由该服务器管理的数据库的清单。例如，如果想要处理由名为 NVBizNet2 的 SQL Server 管理的数据库，可单击 NVBizNet2 图标左边的加号 (+)。
4. 单击包括要使用 FOREIGN KEY 约束关联的表的数据库。对本例来说，单击 SQLTips 数据库图标。Enterprise Manager 将显示默认的数据库图表。如果还没有默认的数据库图表，当 Enterprise Manager 询问是否想要创建数据库图表时，可单击 Yes 按钮。Enterprise Manager 将启动 Create Database Diagram Wizard（创建数据库图表向导）。
5. 如果在步骤 4 中选择的数据库已经有了默认数据库图表（而且 Enterprise Manager 在第 4 步中没有提示启动 Create Database Diagram Wizard），可选择 Action 菜单中的 New 选项并单击 Database Diagram。Enterprise Manager 将启动 Create Database Diagram Wizard。
6. 在 Create Database Diagram Wizard 的欢迎屏幕上，单击 Next 按钮。向导将显示 Select Tables to Be Added（选择要添加的表）对话框，如图 8.2 所示。

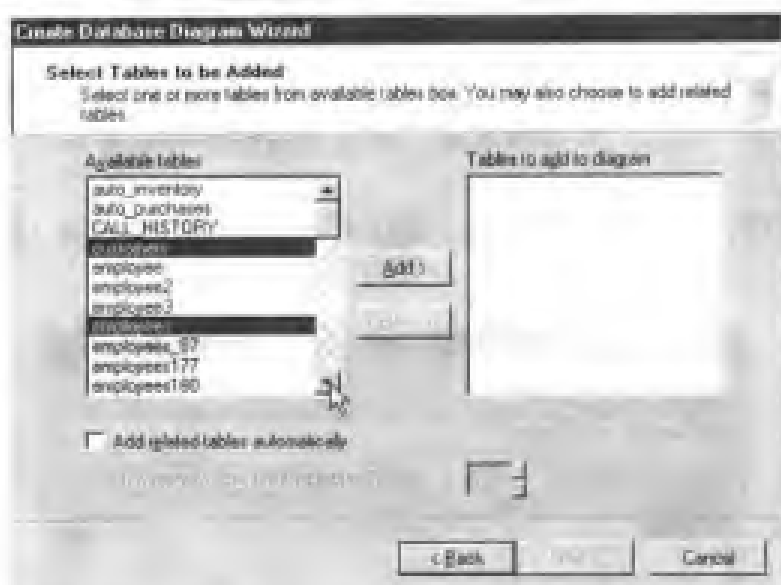


图 8.2 Create Database Diagram Wizard 的 Select Tables to Be Added 对话框

7. 从 Available Tables 的滚动列表中，选择想要建立 FOREIGN KEY 关系的表。对本例来说，按住 Ctrl 键并单击 CUSTOMERS、EMPLOYEES 和 ORDERS 表（表是按字母顺序列出的，可能需要使用滚动条才能显示附加的表名）。

8. 释放 Ctrl 键，然后单击 Add 按钮，将步骤 7 中选择的表移到 Tables to Add to Diagram（添加到图表中的表）滚动窗口中。

9. 单击 Next 按钮。Create Database Diagram Wizard 将显示 Completing the Create Database Diagram Wizard（结束创建数据库图表向导）对话框。

10. 为了根据步骤 7 中选择的表创建数据库图表，可单击 Finish 按钮。

11. 在向导创建了数据库图表之后，它将显示 Tables Have Been Added and Arranged（表已被添加并排列）消息框——单击 OK 按钮。Enterprise Manager 将在如图 8.3 所示的窗口中显示数据库图表中的表。

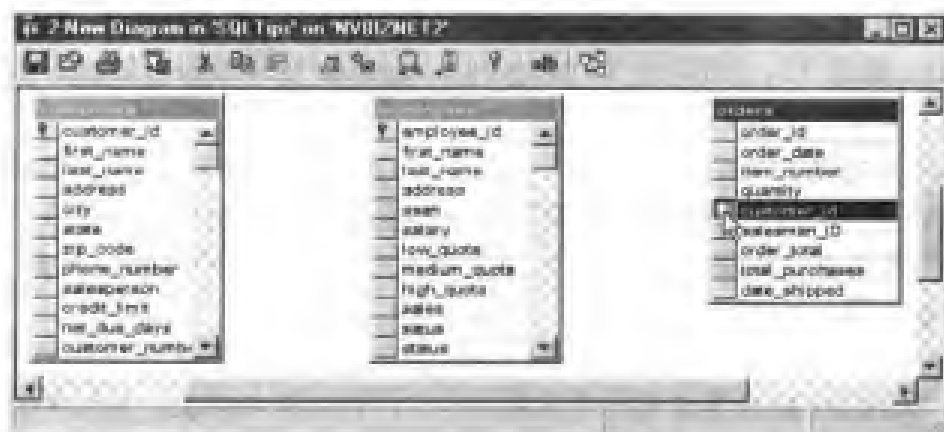


图 8.3 Create Database Diagram Wizard 的 New Diagram 窗口

12. 将鼠标移动到想要用作 FOREIGN KEY 的列左边的选择按钮上，然后按下鼠标左键（如果想要创建多列的 FOREIGN KEY[与多列的 PRIMARY KEY 链接]，可在单击时按住 Ctrl 键。在将鼠标移动到 FOREIGN KEY 的最后一列时按下鼠标左键并释放 Ctrl 键）。对本例来说，

单击 ORDERS 表的 CUSTOMER_ID 列左边的选择按钮。

13. 在按住鼠标左键的同时，拖动鼠标直到经过想要关联 FOREIGN KEY 的表的 PRIMARY KEY 的第一列上。对本例来说，通过将鼠标拖动直到指向 CUSTOMERS 表的 CUSTOMER_ID 列。

14. 释放鼠标左键。Enterprise Manager 将显示 Create Relationship（创建关系）对话框，如图 8.4 所示。

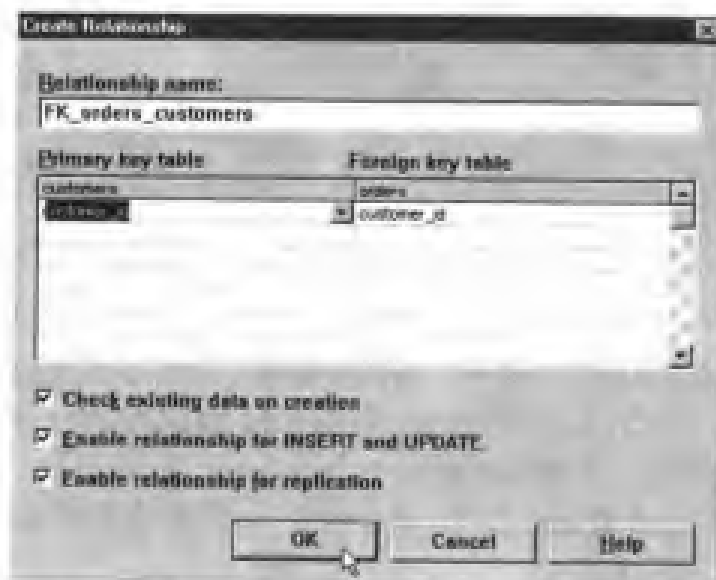


图 8.4 Create Database Diagram Wizard 的 Create Relationship 对话框

15. 单击 OK 按钮，Enterprise Manager 将创建 FOREIGN KEY 并在数据库图表中以图形显示两表间的链接。

16. 对每个想要创建的 FOREIGN KEY 关系，重复步骤 12~步骤 15。对本例来说，在步骤 12 中选择 ORDERS 表中的 SALESMAN_ID 列并在步骤 13 中拖动经过 EMPLOYEES 表的 EMPLOYEE_ID 列。

17. 如果 FOREIGN KEY 链接的图形表重叠，可在 New Diagram 窗口的标准工具栏上单击 Arrange Tables 按钮（右边的按钮），Enterprise Manager 将尽量排列数据库图表中的表，使之不出现重叠，如图 8.5 所示。

在完成创建新 FOREIGN KEY 关系之后，单击 Database Diagram 窗口右上角的关闭按钮（×）退出。Enterprise Manager 将询问是否想要将更新的图表保存为磁盘文件。不管是保存还是不保存，已经创建的 FOREIGN KEY 关系保持不变。对本例来说，单击 NO 按钮返回 Enterprise Manager 窗口。

注意：如果对步骤 4 中创建默认的数据库图表时回答 Yes，不必退出 Database Diagram 窗口。Enterprise Manager 将在其右窗格中继续显示默认的数据库图表，直到单击了另一数据库对象为止。当选择处理另一数据库对象或向导时或退出 Enterprise Manager 时，Enterprise Manager 将询问是否要保存默认的数据库图表。



图 8.5 Create Database Diagram Wizard 的 New Diagram 窗口，
图形化地说明了表间的 FOREIGN KEY 链接

技巧 140 使用 MATCH FULL 子句保持引用完整性

在技巧 134 中，读者了解到，在 FOREIGN KEY 列带有 NULL 值的一行并不违反引用完整性，DBMS 认为，缺少数据的“真正”值将是相关表中的 PRIMARY KEY 值之一。这个“疑问的好处”规则在 FOREIGN KEY 由单列组成时似乎足够清楚了。但是，当表有多列的 FOREIGN KEY 时，组成键字的任何一列中的 NULL 值将使得 DBMS 默认地跳过对 FOREIGN KEY 的引用完整性检查并允许在键字的其余列中的任何 NULL 或合法的非 NULL 值（对于数据类型来说）。

例如，有由以下语句创建的两个表：

```
CREATE TABLE invoice
(cust_ID    INTEGER,
 invoice_num INTEGER)
CREATE TABLE order_detail
(cust_id    INTEGER,
 invoice_num INTEGER,
 order_date DATETIME,
 item_number INTEGER,
CONSTRAINT fk_invoice_order_detail
FOREIGN KEY (cust_ID, invoice_num)
REFERENCES invoice (cust_ID, invoice_num))
```

如果 invoice 表包括带有 CUST_ID 为 1 且 INVOICE_NUM 为 1 的单行，以下 INSERT 语

句明显是合法的：

```
INSERT INTO order_detail VALUES (1,1,07/30/2000,1000)
```

因为新行中复合的 FOREIGN KEY 值 (1,1) 作为 PRIMARY KEY 存在于相关的表中。而以下 INSERT 语句也是合法的：

```
INSERT INTO order_detail VALUES (NULL,23,07/30/2000,1001)
```

```
INSERT INTO order_detail VALUES (77,NULL,07/30/2000,1002)
```

虽然 INVOICE 表没有 INVOICE_NUM 列值为 23 的行，在 CUST_ID 列上也没有 77，每条 INSERT 语句的 FOREIGN KEY (NULL, 23) 和 (27, NULL) 中的 NULL 值告诉 DBMS 挂起对整个 FOREIGN KEY 的引用完整性的评估，而不只是对一列或几列中的 NULL 值。

为了防止 DBMS 允许与对应的 PRIMARY KEY 不匹配的非 NULL 值出现在 FOREIGN KEY 列中，可在 FOREIGN KEY 约束的定义中包括 MATCH FULL 子句。例如，如果在本例中用以下语句创建了 ORDER_DETAIL 表：

```
CREATE TABLE order_detail
(cust_id    INTEGER,
invoice_num INTEGER,
order_date  DATETIME,
item_number INTEGER,
CONSTRAINT fk_invoice_order_detail
FOREIGN KEY (cust_id, invoice_num)
REFERENCES invoice (cust_id, invoice_num) MATCH FULL)
```

FOREIGN KEY 约束上的 MATCH FULL 选项告诉 DBMS，多列 FOREIGN KEY 中的所有的列必须要么为 NULL，要么都为非 NULL。这样，DBMS 将只接受全为 NULL 值或是其非 NULL 复合键值与相关表中的 PRIMARY KEY 中的键字的复合值相匹配(如果 FOREIGN KEY 定义中包括了 MATCH FULL 子句，DBMS 将不允许插入带有部分 NULL 值的 FOREIGN KEY 的行)。

注意：并不是所有的 DBMS 产品（包括 MS-SQL Server）都支持在 FOREIGN KEY 约束的定义中包括 MATCH FULL 子句。如果具体的 DBMS 不支持 MATCH FULL 选项，必须把 NOT NULL 约束放在 FOREIGN KEY 中的每一列，以保证 FOREIGN KEY 列中的所有非 NULL 值与对应的相关表中的 PRIMARY KEY 相匹配。

技巧 141 理解 MATCH FULL、MATCH PARTIAL 和 MATCH SIMPLE 子句

正如读者从技巧 130“理解引用完整性检查和外键”中的引用完整性和 FOREIGN KEY 约束的讨论中所了解的，引用完整性要求，一个表中的 FOREIGN KEY 值要与在 FOREIGN KEY 定义中引用的表的一个且只有一个 PRIMARY KEY 值匹配。因而，引用完整性检查是相对直观的，即检查 PRIMARY KEY，除了在处理部分 NULL 的 FOREIGN KEY 之外确保它有与 FOREIGN KEY 值相匹配的值。正如读者在技巧 140“使用 MATCH FULL 子句保持引用完整性”中所学习的，DBMS 并不对有 NULL 值的 FOREIGN KEY 中的任何列执行引用完整性检查。

SQL-92 提供了 3 种 MATCH 子句，可添加到 FOREIGN KEY 约束的定义中，用来控制当处理 FOREIGN KEY 中的 NULL 值时 DBMS 的行为：

- MATCH FULL. 告诉 DBMS 要求 FOREIGN KEY 中的所有列要么为 NULL 要么为非 NULL。例如，包括 MATCH FULL 子句的两列 FOREIGN KEY 定义告诉 DBMS，键值 (NULL, NULL) 是可接受的。但是，键值如 (100, NULL) 和 (NULL, 100) 都

是违反引用完整性的，因为键字部分为 NULL，部分为非 NULL。

- **MATCH PARTIAL**。告诉 DBMS 允许部分 NULL 的 FOREIGN KEY 值，只要 FOREIGN KEY 中非 NULL 列中的每个与 PRIMARY KEY 的对应列相匹配。这样，包括 MATCH PARTIAL 子句的两列 FOREIGN KEY 约束的定义告诉 DBMS，键值 (NULL, NULL) 是可接受的，但是，键值如 (100, NULL) 和 (NULL, 100)，仅当表中的一行或多行中 PRIMARY KEY 的对应列的值为 100 时，才能通过引用完整性检查。
- **MATCH SIMPLE**。告诉 DBMS 对部分 NULL 的 FOREIGN KEY 值跳过关系完整性检查。因而，如果两列 FOREIGN KEY 的定义包括 MATCH SIMPLE 子句（或完全没有 MATCH 子句），则 DBMS 将接受有值为 (NULL, NULL)、(NULL, 100) 和 (100, NULL) 的 FOREIGN KEY——不管表行中的 PRIMARY KEY 中有没有值为 100 的对应列。

并不是所有的 DBMS 产品都在 FOREIGN KEY 约束的定义中支持 MATCH 子句。因此，请检查系统手册，看一看定义 FOREIGN KEY 约束的句法是否包括以下子句：

```
[MATCH {FULL | PARTIAL | SIMPLE}]
```

如果具体的 DBMS，如 MS-SQL Server 不支持 MATCH 子句，默认的对部分 NULL 的 FOREIGN KEY 值的处理方法将与 MATCH SIMPLE 中描述的一样。

注意：某些 DBMS 产品可能将 MATCH SIMPLE 称为 MATCH UNIQUE。虽然名称不同，但 MATCH UNIQUE 的作用与 MATCH SIMPLE 是一样的。

技巧 142 理解 SET NULL 规则与 MATCH 子句的相互作用

在技巧 137 “理解如何应用 SET NULL 规则更新和删除以帮助保持引用完整性”中，读者了解到，ON DELETE SET NULL 规则告诉 DBMS 在从相关表中删除父行时，在子行的 FOREIGN KEY 列放入 NULL 值。类似地，当用户 ID 改变父行的 PRIMARY KEY 值时，ON UPDATE SET NULL 规则告诉 DBMS 将子行中 FOREIGN KEY 列的值改变为 NULL。如果 FOREIGN KEY 中的一列或多列有 NOT NULL 约束，DBMS 并不报告错误，而是把 NULL 值放入只接受 NULL 值的 FOREIGN KEY 列。

例如，如果有两个由以下语句创建的表：

```
CREATE TABLE customers
(cust_ID      INTEGER,
 office      INTEGER,
 customer_name VARCHAR(30))
CONSTRAINT PK_customers PRIMARY KEY (cust_ID, office)
CREATE TABLE invoices
(cust_ID      INTEGER NOT NULL,
 office      INTEGER,
 invoice_num  INTEGER,
 order_date   DATETIME,
 item_number  INTEGER,
 CONSTRAINT fk_customers_invoices
 FOREIGN KEY (cust_ID, office)
 REFERENCES customers (cust_ID, office))
```

```
ON DELETE SET NULL ON UPDATE SET NULL)
```

并且执行以下 DELETE 语句:

```
DELETE FROM customers WHERE cust_ID = 1 AND office = 5
```

在 INVOICES 表中, DBMS 将把 OFFICE 设置为 NULL, 而把 CUST_ID 为 1 而且 OFFICE 为 5 的所有行的 CUST_ID 列保留为 1。因此, 删除父行将使得 DBMS 在所有的 FOREIGN KEY 列没有 NOT NULL 约束的每个子行中创建部分 NULL 的 FOREIGN KEY 值。

正如读者在技巧 140 中所学的, 创建部分 NULL 值的外键并不是所希望的, 因为 DBMS 跳过对部分 NULL 的键值的引用完整性检查, 而子行的 FOREIGN KEY 列可能保存着并不存在于父表中任何行的对应列的值。

例如, 在本例中, 执行了 DELETE 语句之后, INVOICES 表中的所有子行 (CUST_ID 为 1 且 OFFICE 为 5 的行) 仍然显示 1 号顾客在系统中有发票 (虽然该顾客的 OFFICE 为 NULL)。但是, CUSTOMERS 表 (其中每个顾客应有一行) 将没有 1 号顾客的信息——如果只有 OFFICE 5 有一个顾客的 CUST_ID 为 1。另外, 如果其他办公室确实有 CUST_ID 为 1 的顾客, 这些顾客中的每一个现在是 CUST_ID 为 1 和 NULL 值的子行的父行。

可以向 FOREIGN KEY 中添加 MATCH 子句, 以便告诉 DBMS, 当为违反引用完整性的 DELETE 和 UPDATE 语句应用 ON DELETE NULL 或 ON UPDATE NULL 规则时, 部分 NULL 的 FOREIGN KEY 的创建如何反应。如果在 FOREIGN KEY 的定义中未明确地包括 MATCH 子句, DBMS 使用默认的 MATCH SIMPLE 子句。结果, 当遵循 ON DELETE 或 ON UPDATE SET NULL 规则时, DBMS 被允许创建部分 NULL 的 FOREIGN KEY。

但是, 如果将 INVOICES 表中的 FOREIGN KEY 约束改变为:

```
CONSTRAINT fk_customers_invoices
  FOREIGN KEY (cust_ID, office)
  REFERENCES customers (cust_ID, office)
  MATCH FULL ON DELETE SET NULL ON UPDATE SET NULL)
```

MATCH FULL 子句告诉 DBMS, 不允许创建部分 NULL 的 FOREIGN KEY 值。因而, DBMS 将中止以下 DELETE 语句的执行:

```
DELETE FROM customers WHERE cust_ID = 1 AND office = 5
```

因为它违反了引用完整性约束。ON DELETE SET NULL 规则通常允许 DBMS 成功地执行 DELETE 语句, 从而将子行中的 OFFICE 列设置为 NULL, 而且跳过对所创建的部分 NULL 的 FOREIGN KEY 值的引用完整性检查。但是, MATCH FULL 子句告诉 DBMS 它不能创建部分 NULL 的 FOREIGN KEY 值。

如果想要 DBMS 仅当在每个 FOREIGN KEY 列中的非 NULL 值与父表中一行的对应列相匹配时, 才允许创建部分 NULL 的 FOREIGN KEY 值, 可在 FOREIGN KEY 定义中包括 MATCH PARTIAL 子句。例如, 如果在 INVOICES 表中将 FOREIGN KEY 的定义改为:

```
CONSTRAINT fk_customers_invoices
  FOREIGN KEY (cust_ID, office)
  REFERENCES customers (cust_ID, office)
  MATCH PARTIAL ON DELETE SET NULL ON UPDATE SET NULL)
```

MATCH PARTIAL 子句告诉 DBMS, 它可成功地执行以下 DELETE 语句:

```
DELETE FROM customers WHERE cust_ID = 1 AND office = 5
```

通过遵循 ON DELETE SET NULL 规则, 将 INVOICES 表中的 FOREIGN KEY 的 OFFICE

列设置为 NULL, 而使受 NOT NULL 约束的 CUST_ID 值为 1, 只要 CUSTOMERS 表中至少有一行的 CUST_ID 列值为 1。

技巧 143 使用 NOT NULL 列约束防止列中的 NULL 值

正如人们所了解的, 列中的 NULL 值表明, 列的实际值既可是未知的也可是缺少的。Codd 的 12 条关系数据库定义的准则中的第 3 条准则要求, 每种数据类型的域(合法值的范围)中都包括 NULL 值。但是, 有时想要防止用户向表的某些列中输入 NULL 值。通过向列的定义中添加 NULL 约束, 可告诉 DBMS 中止任何向该列放入 NULL 值的 INSERT 或 UPDATE 语句的执行(如果在执行违反引用完整性的 DELETE 或 UPDATE 语句时必须应用 ON DELETE SET NULL 或 ON UPDATE SET NULL 规则, FOREIGN KEY 列中的 NULL 约束还可防止 DBMS 将列值改变为 NULL)。

在创建新表时, 为了对列应用 NOT NULL 约束, 只要在列的数据类型之后添加该约束。例如, 创建 CUSTOMER 表的以下 CREATE TABLE 语句:

```
CREATE TABLE customer
(cust_ID    INTEGER NOT NULL,
 first_name VARCHAR(30),
 last_name  VARCHAR(30) NOT NULL,
 address    VARCHAR(50) NOT NULL)
```

在这个表中, DBMS 不允许用户向 CUST_ID、LAST_NAME 或 ADDRESS 列中放入 NULL 值。由此, 以下 INSERT 语句将成功执行:

```
INSERT INTO customer VALUES (0, '', '', '')
```

请记住, 值为 0 和长度为 0 的字符串都不是 NULL。但是, 以下 INSERT 语句却不能执行:

```
INSERT INTO customer
VALUES (1, 'Konrad', NULL, '765 E. Eldorado Lane')
```

并出现如下的错误消息:

```
Server: Msg 515, Level 16, State 2, Line 1
Cannot insert the value NULL into column 'last_name', table
'SQLTips.dbo.customer'; column does not allow nulls.
INSERT fails.
The statement has been terminated.
```

因为 INSERT 语句试图添加 LAST_NAME 列为 NULL 的行, 这个列在表定义中是受约束的且有 NOT NULL 约束。类似地, DBMS 将中止以下 UPDATE 语句的执行:

```
UPDATE customer SET address = NULL WHERE cust_ID = 0
```

因为此语句试图向表中已有行的受 NOT NULL 约束的列中放入 NULL 值。

某些 DBMS 产品仅当第一次创建时才允许指定一系列的 NOT NULL 约束。但是, MS-SQL Server 允许使用 ALTER TABLE 语句向已有表中的一列上添加 NOT NULL 约束——即使该表中有数据。例如, 以下 ALTER TABLE 语句将向 CUSTOMER 表的 FIRST_NAME 列添加 NOT NULL 约束:

```
ALTER TABLE customer
ALTER COLUMN first_name VARCHAR(30) NOT NULL
```

如果 CUSTOMER 表中的任何一个已有行在 FIRST_NAME 列中有 NULL 值, ALTER TABLE 语句将中止, 出现与前面的试图向受 NOT NULL 约束的列中放入 NULL 值的 INSERT

语句所显示的类似的错误消息。为了解决这一问题，可用以下 UPDATE 语句将该列中的 NULL 值改为非 NULL 值，然后再执行 ALTER TABLE 语句：

```
UPDATE customer SET first_name = ''
WHERE first_name IS NULL
```

技巧 144 使用 UNIQUE 列约束防止列中的重复值

正如读者在技巧 129 “使用 PRIMARY KEY 列约束惟一地确定表行”中所学过的，可以使用 PRIMARY KEY 约束告诉 DBMS 强制列中的值在表中的每一行上都是不同的（惟一的）。遗憾的是，表只能有一个 PRIMARY KEY，而人们常常要求在每一行上不止一列包括惟一值。

例如，假设每个雇员必须有惟一的 ID（为了标识身份），而且安全负责人还要求每个雇员有惟一的密码。此时可使用以下 CREATE TABLE 语句创建带有多个 UNIQUE 列约束的表：

```
CREATE TABLE employees
(
  ID          INTEGER PRIMARY KEY,
  name        VARCHAR(30),
  SSAN        INTEGER UNIQUE,
  password    VARCHAR(15) UNIQUE)

```

在本例中，PRIMARY KEY 约束指定，ID 列必须包括惟一的、非 NULL 的值。另外对 SSAN 和 password（密码）列的 UNIQUE 约束告诉 DBMS 防止向这些列中输入重复的值。

```
INSERT into employees
VALUES (1, 'Konrad King', 123456789, 'SECRET')
```

只要在 EMPLOYEES 表不存在一行，其中或 ID 为 1，或 SSAN 值为 123456789，或密码为 SECRET，则上面语句可成功执行。如果一条 INSERT 语句试图向受 UNIQUE 约束的任一列或多列中添加重复值，则 DBMS 将中止其执行，并显示如下错误消息：

```
Server: Msg 2627, Level 14, State 2, Line 1
Violation of UNIQUE KEY constraint 'UQ_employees_1D4655FB'.
cannot insert duplicate key in object 'employees'.
The statement has been terminated.
```

在执行 UPDATE 语句时，DBMS 还实施 UNIQUE 约束，例如，以下 UPDATE 语句仅当 EMPLOYEES 表中没有行在 PASSWORD 列中的值为 NEW PASSWORD 时，才能成功执行：

```
UPDATE employees SET password = 'NEW PASSWORD' WHERE ID=1
```

否则，DBMS 将中止，并显示与向受 UNIQUE 约束的列中添加重复值时类似的错误消息。

注意：用 PRIMARY KE 约束定义的列与受 UNIQUE 约束的列之间有一个重要的差别。DBMS 将允许向 UNIQUE 列插入单个 NULL 值。相反，在 PRIMARY KEY 列中 NULL 值是不允许的。

技巧 145 使用 CHECK 约束确认列值

CHECK 约束是可用于将表中的一列或多列的域（或合法值的范围）限制为该数据类型全部允许值范围内的子集的工具。每个 CHECK 约束由单词 CHECK 后跟搜索条件组成，DBMS 每当执行 DELETE、INSERT 或 UPDATE 语句改变表的内容时都对这些搜索条件求值。如果修改后搜索条件求值为 TRUE，则 DBMS 允许该修改；如果搜索条件求值为 FALSE，系统则取消所完成的工作并返回一条错误。

CHECK 约束的句法如下:

```
[CONSTRAINT <constraint name>] CHECK (<search condition>)
```

因而,可在以下 CREATE TABLE 语句中使用 CHECK 约束来创建 EMPLOYEES 表,DBMS 将检查由 INSERT 或 UPDATE 语句提供的 DEPARTMENT 值,并确保其处于 CHECK 约束的搜索条件所列出的范围内:

```
CREATE TABLE employees
  (ID          INTEGER PRIMARY KEY,
   name        VARCHAR(30) NOT NULL,
   SSAN        INTEGER UNIQUE NOT NULL,
   department  VARCHAR(15) CONSTRAINT valid_department
   CHECK (department IN ('MARKETING','SALES','ADMIN',
                        'EXECUTIVE','SERVICE',
                        'COLLECTIONS','WAREHOUSE')),
   hourly_rate MONEY,
   monthly_salary MONEY,
   commission  MONEY)
```

在本例中, DBMS 将允许使用以下 INSERT 语句向表中添加行:

```
INSERT INTO employees VALUES
  (1,'Konrad King',123456789,'Executive',NULL,3000.00,NULL)
```

但将拒绝以下 UPDATE 语句的执行:

```
UPDATE employees SET department = 'MIS' WHERE ID = 1
```

并显示如下错误信息:

```
Server: Msg 547, Level 16, State 1, Line 1
UPDATE statement conflicted with COLUMN CHECK constraint
'valid_department'. The conflict occurred in database
'SQLTips', table 'employees'.
The statement has been terminated
```

除了检查以确保单列中的数据必须落在范围之内或必须是离散值集合中的一个之外,还可将 CHECK 约束用于检查多列中数据的合法性,甚至可用一列中的值来定义另外列的域。例如,以下 CREATE TABLE 语句:

```
CREATE TABLE employees
  (ID          INTEGER PRIMARY KEY,
   name        VARCHAR(30) NOT NULL,
   SSAN        INTEGER UNIQUE NOT NULL,
   department  VARCHAR(15) CONSTRAINT valid_department
   CHECK (department IN ('MARKETING','SALES','ADMIN',
                        'EXECUTIVE','SERVICE',
                        'COLLECTIONS','WAREHOUSE')),
   hourly_rate MONEY,
   monthly_salary MONEY,
   commission  MONEY,
  CONSTRAINT valid_pay CHECK(
    (department IN ('MARKETING','SALES') AND
     (hourly_rate IS NULL) AND (monthly_salary IS NULL) AND
     (commission BETWEEN 100.00 and 500.00)) OR
    (department IN ('ADMIN','EXECUTIVE') AND
```

```
(hourly_rate IS NULL) AND (monthly_salary BETWEEN  
1000.00 AND 10000.00) AND (commission IS NULL)) OR  
(department IN ('SERVICE', 'COLLECTIONS', 'WAREHOUSE') AND  
(hourly_rate BETWEEN 10.00 AND 45.00) AND  
(monthly_salary IS NULL) AND (COMMISSION IS NULL)))
```

告诉 DBMS 确保 EMPLOYEES 表满足以下条件：

- 为 MARKETING 和 SALES 中的雇员指定佣金（每次销售）处于\$100.00 和\$500.00 之间，而且没有每月的工资或每小时的工资。
- 为 ADMIN 和 EXECUTIVE 中的雇员指定每月工资处于\$1,000.00 和\$10,000.00 之间，而没有每小时的工资和佣金。
- SERVICE、COLLECTIONS 和 WAREHOUSE 中的雇员每小时的工资数处于\$10.00 和\$45.00 之间，而没有月工资和佣金。

技巧 146 使用 MS-SQL Server Enterprise Manager 将规则与数据类型或列绑定在一起

规则是可用于指定列或数据类型的域的 MS-SQL Server 对象。MS-SQL Server 在进行 CHECK 约束检查时使用规则。每当 SQL 语句将值放入由规则支配的列中时，DBMS 都要对规则的条件表达式求值。如果表达式求值为 TRUE，DBMS 就允许更新；如果规则的条件表达式求值为 FALSE，MS-SQL Server 取消所完成的工作并报告错误。

与 CHECK 约束不同，规则是数据库对象，而且不是表或列定义的一部分。因此，在 DBMS 应用规则来检查要放入列中的数据的合法性之前，必须首先将规则与列或用于定义列的用户定义的数据类型绑定在一起。为了将规则与列或用于定义列的用户定义的数据类型绑定在一起既可执行 sp_bindrule 存储过程（将在技巧 466 中学习）也可以执行以下步骤，使用 MS-SQL Server Enterprise Manager 来达到目的：

1. 为了启动 Enterprise Manager，可单击 Start 按钮，将鼠标指针移动到 Start 菜单的 Programs 上并选择 Microsoft SQL Server 7.0，然后单击 Enterprise Manager。
2. 为了显示 SQL 服务器的列表，可单击 SQL Server Group 左边的加号 (+)。
3. 找出管理着想要将规则与表列或数据类型绑定在一起的数据库的 SQL 服务器，单击 SQL Server 图标左边的加号 (+)。例如，如果想要处理由名为 NVBizNet2 的 SQL Server 管理的数据库，可单击 NVBizNet2 图标左边的加号 (+)。Enterprise Manager 将显示由该服务器管理的数据库的清单。
4. 单击包括要定义规则的数据库。对本例来说，单击 SQLTips 数据库图标左边的加号 (+)。Enterprise Manager 将显示该数据库中的每个对象类的图标。
5. 在展开的数据库对象的清单中单击 Rules（规则）。Enterprise Manager 将在其右窗格中显示在此数据库中定义的规则的清单。
6. 双击想要绑定的规则。对本例来说，双击 VALIDATE_EMPNUM 规则。Enterprise Manager 将显示 Rule Properties（规则属性）对话框，如图 8.6 所示。
7. 如果想要将步骤 6 中选定的规则与用户定义的数据类型绑定在一起，可从步骤 11 处继续。否则为，将规则与列绑定在一起，单击 Bind Columns（绑定列）按钮。Enterprise Manager 将显示 Bind Rule to Columns（将规则与列绑定）对话框，如图 8.7 所示。

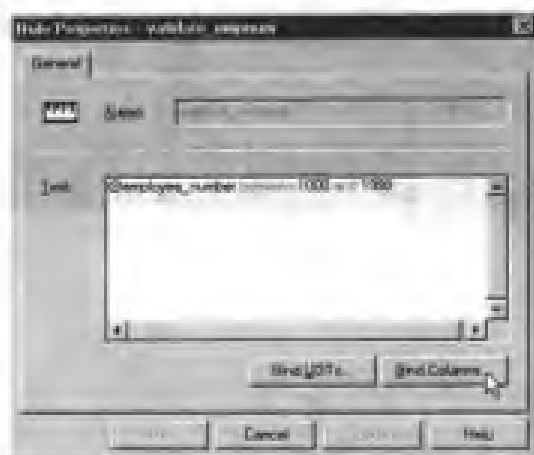


图 8.6 Enterprise Manager 的 Rule Properties 对话框



图 8.7 Enterprise Manager 的 Bind Rule to Columns 对话框

8. 为了选择想要将规则与列绑定的表，可单击该对话框的 Table 域右边的下拉按钮。然后从数据库表的下拉列表中选择表。对本例来说，单击数据库清单中的[dbo].[employees]选择 EMPLOYEES 表 Enterprise Manager 将在 Unbound Columns（未绑定的列）框中显示还未与规则绑定的列。

9. 为了将规则与列绑定，在 Unbound Columns 框中单击想要绑定的列名。接着单击 Add 按钮。对本例来说，单击 EMPLOYEE_ID 列，然后单击 Add 按钮。Enterprise Manager 将所选列名从 Unbound Columns 清单中转移到 Bound Columns（绑定的列）清单中。

10. 如果想要将规则与另外的列绑定，可重复步骤 8 和步骤 9。完成绑定后，单击 OK 按钮。Enterprise Manager 接着显示 Rule Properties 对话框。

11. 如果想要将规则与用户定义的数据类型绑定，则单击 Bind UDTs 按钮。Enterprise Manager 显示 Bind Rule to User-Defined Data Types（将规则与用户定义的数据类型绑定）对话框，如图 8.8 所示。如果不想将规则与用户定义的数据类型绑定，可从步骤 15 处继续。



图 8.8 Enterprise Manager 的 Bind Rule to User-Defined Data Types 对话框

12. 为了将规则与数据类型绑定，可单击想要绑定的数据类型行的 Bind 复选框。对本例来说，单击 VALID_EMPNUMS 数据类型使之出现选择标记。如果想要此后规则只应用于使用该数据类型定义的列，可单击 Future Only 复选框，否则，清除该复选框，DBMS 将把规则应用于使用该数据类型创建的列。对本例来说，清除 Future Only 复选框。

13. 如果想要将规则与另外的用户定义的数据类型绑定，可重复步骤 12，直到完成。

14. 为了退出对话框，单击 OK 按钮。Enterprise Manager 将返回 Rule Properties 对话框。

15. 为了退出 Rule Properties 对话框并返回 Enterprise Manager 窗口，单击 OK 按钮。

MS-SQL Server 只允许对数据库表中的每一列绑定一条规则。但是，可将一条规则及一个或多个 CHECK 约束与同一列绑定。另外，DBMS 仅当对绑定的列（或使用与规则绑定的数据类型定义的列）的值改变时才应用规则。DBMS 并不检查规则绑定的列的已有值，从而确保其不违反规则。

技巧 147 使用 Transact-SQL 的 CREATE RULE 语句创建 MS-SQL Server 规则

创建规则既可使用 Enterprise Manager，也可使用 Transact-SQL 的 CREATE RULE 语句。

CREATE RULE 语句的句法如下：

```
CREATE RULE <rule name>
```

```
AS @<parameter> <conditional predicate>
```

因而，为了创建将范围为 1000.00~10000.00 内的值放入列的规则，可执行以下 CREATE RULE 语句：

```
CREATE RULE validate_salary
```

```
AS @salary BETWEEN 1000.00 AND 10000.00
```

或者，为了创建一条规则，告诉 DBMS 确保放入列内的值是列表中的一个，可执行以下 CREATE RULE 语句：

```
CREATE RULE validate_freshman_classes
```

```
AS @class IN ('English','History','Math','Science')
```

CREATE RULE 语句只在数据库中创建一个规则对象。在 DBMS 用此规则检查要改变的

或要添加的列值之前,必须使用 Enterprise Manager (正如读者在技巧 146 中所学习的)或是执行 `sp_bindrule` 存储过程,将规则与列或其用户定义的数据类型绑定在一起。

执行 `sp_bindrule` 存储过程的句法如下:

```
EXEC '<rule name>',  
    '<table name>.<column name>',['FUTUREONLY']
```

这样一来,如果要将 `VALIDATE_SALARY` 规则与 `EMPLOYEES` 表的 `MONTHLY_SALARY` 列绑定,可键入以下 Transact-SQL 语句:

```
EXEC sp_bindrule  
    'validate_salary','employees.monthly_salary'
```

技巧 148 使用 MS-SQL Server Enterprise Manager 的 Rule Properties 屏幕改变规则

正如读者在技巧 145~技巧 147 中所学习的,MS-SQL Server 规则和 SQL 的 CHECK 约束允许 DBMS 对用户或应用程序试图向表行中的列放入的值执行合法性测试。虽然 CHECK 约束作为表的定义的惟一元素存在,但每条 MS-SQL Server 规则是独立的数据库对象而不是列或用户定义的数据类型定义的一部分。因而,CREATE TABLE 和 ALTER TABLE 语句中的子句管理 CHECK 约束,而 MS-SQL Server 提供一套存储过程和 Enterprise Manager 来管理规则,正如管理其他数据库对象一样。

为了使用 MS-SQL Server Enterprise Manager 处理规则属性,可执行以下步骤:

1. 为了启动 Enterprise Manager,可单击 Start 按钮,将鼠标指针移动到 Start 菜单的 Programs 上并选择 Microsoft SQL Server 7.0,然后单击 Enterprise Manager。
2. 为了显示 SQL 服务器的列表,可单击 SQL Server Group 左边的加号 (+)。
3. 找出管理着想要改变的数据库的 SQL 服务器,单击 SQL Server 图标左边的加号 (+)。例如,如果想要处理由名为 NVBizNet2 的 SQL Server 管理的数据库,可单击 NVBizNet2 图标左边的加号 (+)。Enterprise Manager 将显示由该服务器管理的数据库的清单。
4. 单击包括要改变规则的数据库左边的加号 (+)。对本例来说,单击 SQLTips 数据库图标左边的加号 (+)。Enterprise Manager 将显示该数据库中的每个对象类的图标。
5. 在展开的数据库对象的清单中单击 Rules (规则)。Enterprise Manager 将在其右窗格中显示在此数据库中定义的规则的清单。
6. 为了处理规则的属性,右击该规则 (或其图标)。对本例来说,单击 `VALIDATE_FRESHMAN_CLASSES`。Enterprise Manager 将显示规则属性对话框,如图 8.9 所示。
7. 为了改变允许 DBMS 放入一列的值范围,请在该对话框的 Text 域内修改规则的搜索条件。对本例来说,假定新生班级名包括一个 -101 后缀,而且工程课程已经添加到课程表中。因此,用 `@class(English-101,'History-101','Math-101','Science-101','Engineering-101')` 取代 Text 框中的内容。
8. 单击 OK 按钮。Enterprise Manager 将保存新规则的定义,并关闭 Rule Properties 对话框。

使用规则而不使用 CHECK 约束的主要好处之一是,虽然 CHECK 约束可确认只对一列的更新,而每条规则可对将要放入一个表或多个表中的多列中的数据加以确认。由此,如果同一 CHECK 约束用于几个表中,必须对每一个表都执行 ALTER TABLE 语句,以反映出域的变化

（如在前面的步骤 7 中的一个）。作为对照，如果将单条规则与多列绑定（既可直接绑定也可非直接地将规则与用于定义列的用户定义数据类型绑定），对规则的搜索条件的单条改变（如在步骤 7 中所执行的）将改变与规则绑定的所有列更新的合法性测试。

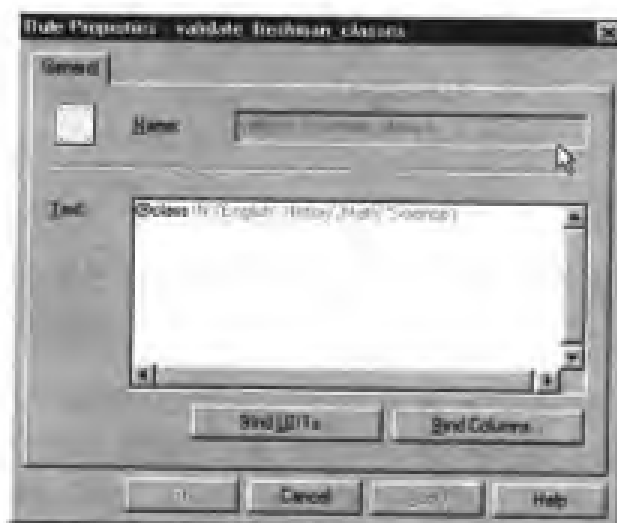


图 8.9 Enterprise Manager 的 Rule Properties 对话框

技巧 149 使用 Transact-SQL 的 DROP RULE 语句永久地从数据库中删除规则

虽然 MS-SQL Server 的规则执行与 SQL 的 CHECK 约束有同样的基本功能，但与 SQL 的 CHECK 约束不同，规则作为独立的数据库对象存在。由于每条 CHECK 约束是表定义的一部分，因而可使用带 DROP <constraint> 子句的 ALTER TABLE 语句删除 CHECK 约束。作为对照，由于规则是数据库对象，可使用 DROP 语句删除规则——就像从数据库中删除任何其他对象一样。

DROP RULE 语句的句法如下：

```
DROP RULE <rule name> [...<last rule name>]
```

由此，可使用单条 DROP RULE 语句从数据库中删除一条或多条规则。例如，为了从数据库中删除 VALIDATE_GRADES 和 VALIDATE_SALARY 规则，可执行以下 DROP RULE 语句：

```
DROP RULE validate_grades, validate_salary
```

在删除数据库规则之前，必须解除规则与列或用户定义的数据类型的绑定。如果试图删除（DROP）一条仍然与列或用户定义的数据类型绑定的规则，MS-SQL Server 将中止 DROP RULE 语句的执行并出现错误消息。遗憾的是，出现的错误信息并未告知规则与之绑定的列或用户定义的数据类型。因此，必须参考数据字典，其中列出了所有的数据库对象及其依赖性（包括规则绑定）。

注意：如果数据字典不可用或过了时，还可使用 Enterprise Manager 的 Bind Rule to Columns 和 Bind Rule to User-Defined Data Types 对话框（已在技巧 146 “使用 MS-SQL Server Enterprise Manager 将规则与数据类型或列绑定在一起”中学习过该内容）来显示（并删除）规则的绑定关系。

在技巧 146 中的步骤 8 中，读者学习了如何选择其中有与列绑定的规则的表。如果一个接一个地选择每个表，Enterprise Manager 将在 Bind Rule to Columns 对话框的 Bound Columns 域内显示与规则绑定的列。为了解除绑定，可单击 Bound Columns 域中的列名，然后再单击 Remove

按钮。如果解除规则与列的绑定，在转到下一表之前，一定要单击 Apply 按钮。

类似地，当执行技巧 146 中的步骤 11 时，Enterprise Manager 将使用 Bind Rule to User-Defined Data Types 对话框列出所有用户定义的数据类型。每个与规则绑定的用户定义的数据类型在其 Bind 复选框上都有选择标记。为了解除绑定，只要单击该复选框清除其选择标记即可。如果清除了任何 Bind 复选框，一定要单击 OK 按钮（不要单击 Cancel 按钮）来退出对话框。

技巧 150 使用 MS-SQL Server Enterprise Manager 列出并编辑视图

视图是虚拟的表。虽然其与“真实”表的外观和感受是一样的，而且可用在大多数 SQL 语句中，只要表引用是允许的，但视图没有其自己的数据行和列，其行和列是由从其他表（或视图）中提取并显示数据的 SELECT 语句组成的。虽然不是“真实的”表，但视图是真正的数据库对象，与视图所基于的表是分开存在的。

为了使用 MS-SQL Server Enterprise Manager 列出并编辑视图，请执行以下步骤：

1. 为了启动 Enterprise Manager，可单击 Start 按钮，将鼠标指针移动到 Start 菜单的 Programs 上并选择 Microsoft SQL Server 7.0，然后单击 Enterprise Manager。
2. 为了显示 SQL 服务器的列表，可单击 SQL Server Group 左边的加号 (+)。
3. 找出管理着想要处理视图的数据库的 SQL 服务器，单击 SQL Server 图标左边的加号 (+)。例如，如果想要处理名为 NVBizNet2 的 SQL Server 管理的数据库，可单击 NVBizNet2 图标左边的加号 (+)。Enterprise Manager 将显示由该服务器管理的数据库的清单。
4. 单击包括想要的视图的数据库左边的加号 (+)。对本例来说，单击 SQLTips 数据库图标左边的加号 (+)。Enterprise Manager 将显示该数据库中的每个对象类的图标。
5. 在展开的数据库对象的清单中单击 Views，Enterprise Manager 将在其右窗格中显示在此数据库中视图的清单。
6. 为了处理视图，可双击视图名（在右窗格中）。对本例来说，双击 CONSTRAINT_COLUMN_USAGE 视图名。Enterprise Manager 将显示 View Properties（视图属性）对话框，如图 8.10 所示。

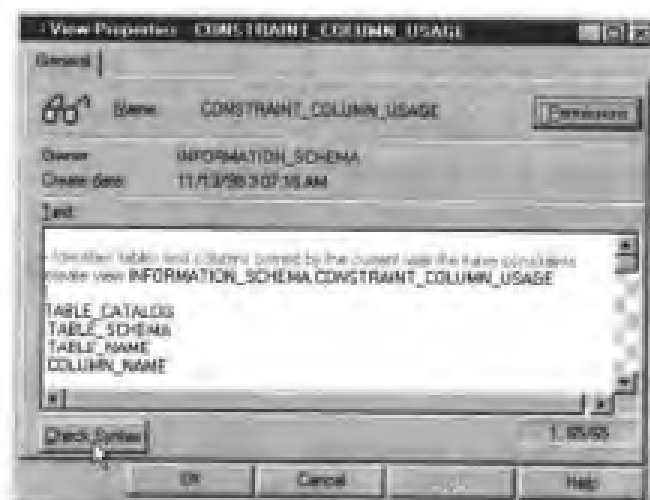


图 8.10 Enterprise Manager 的 View Properties 对话框

7. 为了改变步骤 6 中选择的视图中所显示的数据，可编辑 View Properties 对话框下半部

分的 Text 域中的内容。一定要单击 Check Syntax（检查句法）按钮，以便让 Enterprise Manager 在向磁盘上保存视图的新定义时检查视图的句法，然后单击 OK 按钮。

8. 为了指定用户 ID 和角色对视图的访问权限，可单击靠近对话框右上角的 PERMISSIONS 按钮。Enterprise Manager 将列出数据库用户 ID 和角色，同时列出了显示每个人对视图有的权限的复选框，如图 8.11 所示。

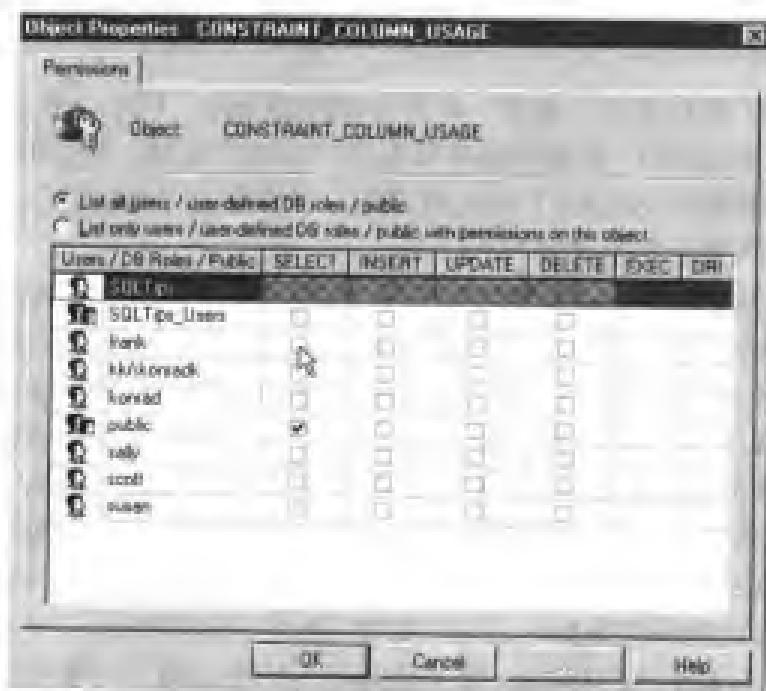


图 8.11 Enterprise Manager 的 Object Properties 屏幕

9. 为了授予用户 ID 或角色以权限，单击用户 ID 或角色行上的想要授予的权限的复选框。例如，为了向用户 ID FRANK 授予 GRANT SELECT 权限，可单击 FRANK 行上的 SELECT 列的复选框，使之出现选择标记。

10. 类似地，为了撤消（REVOKE）对视图的权限，可清除所指的复选框。

11. 在完成授予和撤消对视图的权限之后，单击 OK 按钮并返回 View Properties 对话框。

12. 为了保存对视图所做的改变并返回 Enterprise Manager 窗口，可单击对话框的 OK 按钮。

除了 Enterprise Manager 之外，还可使用 SQL 语句来改变视图及用户 ID 和角色对视图的权限。虽然 GRANT 和 REVOKE 语句与 Enterprise Manager 一样使用方便，但是，ALTER VIEW 语句要求重新输入整个视图的定义——即使只修改视图的一部分也是如此。另一方面，Enterprise Manager 允许编辑想要改变的视图定义的一部分。因此，很明显，Enterprise Manager 是对长的视图定义做出较小改变时的最佳选择。

技巧 151 使用 CREATE ASSERTION 语句创建多表约束

CHECK 约束和 MS-SQL Server 规则限制了 SQL 语句可放入表列中的值。同时断言（assertion）对 DBMS 可在数据库中保存的值作为整体加以限制。

当存在 CHECK 约束时，它包括在表的定义中。规则虽然是独立的数据库对象，但在用于确认 UPDATE 和 INSERT 语句的行动之前，必须与特定的表列或用户定义的数据类型绑定在一起。

断言与规则类似，是独立的对象。但是，与规则不同，它不必与特定的数据类型或列加以绑定，而是要求 DBMS 每当试图改变断言中 CHECK 子句指定的任何表的内容时对断言中的搜索条件求值。

例如，给定 CREATE ASSERTION 语句的以下句法：

```
CREATE ASSERTION <assertion name>  
CHECK (<search condition>)
```

即可创建断言，从而可通过以下 CREATE ASSERTION 语句确保突出的定单中的货品总数不要超过存货目录中的量：

```
CREATE ASSERTION item_in_stock  
CHECK((orders.item_number = inventory.item_number) AND  
      (SUM (orders.qty) <= inventory.qty_on_hand))
```

在向数据库定义中添加了 ITEM_IN_STOCK 断言之后（通过执行本例中的 CREATE ASSERTION 语句），就要求 DBMS 每当 SQL 语句试图改变 ORDERS 或 INVENTORY 表时，对断言中的搜索条件求值。如果更新引起断言求值为 FALSE，DBMS 就取消所完成的工作，生成一条错误并显示错误信息。其中包括由于违反断言而不能执行的语句。

注意：虽然断言包括在 SQL-92 规范中，但大多数数据库产品并不支持断言。因而，在设计数据库中包括断言之前，一定要检查系统手册，看有没有 CREATE ASSERTION 语句。

第 9 章 执行多表查询并创建 SQL 视图

技巧 152 使用带 FROM 子句的 SELECT 语句进行多表查询

除了显示从单个表中的信息之外，还可用 SELECT 语句组合并显示来自多个表中的信息。多表的 SELECT 语句与单表的类似，允许在 SELECT 子句中列出想要看到的列并要求在 FROM 子句中列出 DBMS 要从中提取列值的表（或几个表）。但是，虽然单表 SELECT 语句在没有 WHERE 子句的情况下将显示表中所有行上选定的列值，但多表 SELECT 子句要求一个 WHERE 子句才能生成同样的结果。

当 DBMS 要从两个表（或两个以上的表）中提取数据时，必须包括带有相等条件表达式的 WHERE 子句，表达式告诉 DBMS，在 SELECT 语句的 FROM 子句中列出的每个表对中，一个表中的哪一行要与另一表中的哪一行相匹配，例如，假如有用以下语句创建的表：

```
CREATE customers
(cust_ID INTEGER,
 name      VARCHAR(30),
 address   VARCHAR(50),
 salesrep  INTEGER)

CREATE employees
(salesrep_ID INTEGER,
 name          VARCHAR(30))
```

必须执行以下 SELECT 语句：

```
SELECT * FROM customers, employees
WHERE salesrep = salesrep_ID
```

来生成带有顾客的销售人员姓名和 ID，然后列出 CUSTOMERS 表中的每位顾客的信息。

在告诉 DBMS 一个表的哪一行要与另一表中的哪一行组合起来之后，可包括附加的条件表达式来准确指定想要显示哪一行。例如，为了显示来自 CUSTOMERS 和 EMPLOYEES 表中的那些特定顾客，可使用如下 SELECT 语句：

```
SELECT cust_ID AS 'ID', customers.name, address,
       employees.name AS 'salesperson'
FROM customers, employees
WHERE salesrep = salesrep_ID AND customers.name = 'Jones'
```

除了多表行匹配的（相等）条件表达式之外，其中还包括 CUSTOMERS 表的选择标准。类似地，如果想要根据 SELECT 语句的 FROM 子句中的“其他”表来限制结果表中的行，可执行以下 SELECT 语句：

```
SELECT cust_ID AS 'ID', customers.name, address,
       employees.name AS 'salesperson'
FROM customers, employees
WHERE salesrep = salesrep_ID AND employees.name = 'Smith'
```

这个语句根据销售人员的名字与顾客的姓名来决定显示哪一行。

有关执行多表 SELECT 语句要记住的重要事情是，FROM 子句必须列出所有要显示列值的表名。而且除非想要生成笛卡尔积（或 CROSS JOIN，有关这一内容将在技巧 223 “使用 CROSS JOIN 创建笛卡尔积”中学习），多表 SELECT 语句必须对 FROM 子句中列出的每个表对包括带相等条件表达式的 WHERE 子句（相等条件表达式告诉 DBMS，在将一个表中的一行与另一表中的对应行匹配时使用哪一列的值）。

技巧 153 使用视图显示一个或多个表或视图中的列

正如读者在技巧 8 “理解视图”中所学习的，视图是虚拟表。虽然几乎可在任何允许表引用的地方使用视图，但视图不是与数据库中的其他表和索引一样驻留在硬盘上的物理表，视图是由从一个或多个基表（或其他视图）的行和列中提取数据的 SELECT 语句组成的。当 SQL 语句引用一个视图时，DBMS 执行的 SELECT 语句的结果表就是内存中的表，可用作其他 SQL 语句的目标。

创建视图的句法如下：

```
CREATE VIEW <view name>
[(<column name>[,...<last column name>])]
AS <SELECT statement>
```

这样一来，为了创建基于以下语句所创建表的学生信息的视图：

```
CREATE TABLE students
(SID          INTEGER,
first_name    VARCHAR(15),
last_name     VARCHAR(20),
SSAN         CHAR(11),
home_address  VARCHAR(50),
home_city     VARCHAR(20),
home_state    CHAR(2),
home_phone_number CHAR(14),
major         VARCHAR(20))
```

可使用如下 CREATE VIEW 语句：

```
CREATE VIEW vw_student_list AS SELECT * FROM students
```

而且接着视图可用以下 SELECT 语句来显示学生的信息：

```
SELECT SID AS 'student ID', first_name, last_name
FROM vw_student_list
```

如果考虑为让 DBMS 查询底层表来创建内存内的虚拟表的附加开销，人们会发现没有几个（如果有的话）单视图定义只有 SELECT * 而没有 WHERE 子句。毕竟，如果要显示单表中的所有列和所有行还创建视图干什么？如果一个用户有权访问表中的所有数据，应该避免让 DBMS 生成视图的开销，而让用户对该表（本例中是 STUDENTS 表）执行 SQL 语句更好一些。

使用视图的真正的好处之一是其隐藏不想让其他用户看到的部分表的能力。例如，假设想让用户只能处理那些位于 STUDENTS 表中的 SID、FIRST_NAME 和 LAST_NAME 列中的内容。通过使用以下 CREATE VIEW 语句创建视图：

```
CREATE VIEW vw_student_name_list
(student_ID, first_name, last_name)
```

```
AS SELECT SID, first_name, last_name FROM students
```

以下 SELECT 语句：

```
SELECT * FROM vw_student_name_list
```

将只显示那些底层表中想要用户看到的列。通过授予用户访问视图而不授予访问底层表的权限，可把用户限制为只能处理并显示某些表列。

技巧 154 使用视图显示一个或多个表的特定行中的列

在技巧 153 “使用视图显示一个或多个表或视图中的列”中，读者已经学习了如何通过创建只包括想要用户看到的列的视图来隐藏表中的列。通过授予用户以访问视图而不是底层表的权限，就可把用户限制为在对执行了 SELECT * 语句之后只能看到视图中包括的那些列。

只显示底层表中某些列的视图叫做纵向视图。名称中使用了术语“纵向”，因为当查看表中的数据时，是从一行到另一行按纵向（上或下）进行的。由于表是由纵向的列和水平的行组成的，因而可得出结论，如果有纵向视图也应该有水平视图。

水平视图是允许用户只显示并处理表中的特定行的视图。例如，由以下语句创建的视图就是水平视图：

```
CREATE VIEW vw_LVNV_student_list
AS SELECT * FROM students
WHERE home_city = 'Las Vegas' AND home_state = 'NV'
```

因为用户只能处理 STUDENTS 表中的某些行——即那些来自 Las Vegas（拉斯维加斯）和 Nevada（内华达）的那些学生。本例说明通过在视图的 SELECT 语句中包括 WHERE 子句可创建水平视图。

另外，还可将对视图的纵向和水平限制结合起来，只允许用户看到一个或多个表中的当前行中的某些列。例如，为了将用户限制为只能处理来自 Las Vegas、Nevada 的学生的 ID 和姓名信息，可执行以下 CREATE VIEW 语句：

```
CREATE VIEW vw_LVNV_student_name_list
(student_ID, first_name, last_name)
AS SELECT SID, first_name, last_name FROM students
WHERE home_city = 'Las Vegas' AND home_state = 'NV'
```

除了限制用户只能显示特定的列和行之外，视图还可用于简化用户执行 SELECT 语句从几个表的组合中显示信息。例如，假设除了技巧 153 中的 STUDENTS 之外，还有由以下语句创建的表：

```
CREATE TABLE grades
(class          VARCHAR(15),
section        SMALLINT,
grade          VARCHAR(4),
student_ID     INTEGER,
professor_ID   INTEGER)

CREATE TABLE teachers
(pid           INTEGER,
professor     VARCHAR(30),
department    VARCHAR(20))
```

然后就可使用以下语句创建组合了来自 3 个表中数据的视图：

```
CREATE VIEW vw_students_grades_teachers AS
SELECT * FROM students, grades, teachers
WHERE (grades.student_id = students.SID) AND
      (grades.professor_ID = teachers.PID)
```

在授予对视图的 SELECT 访问权限之后，一个用户可执行以下单表的 SELECT 语句来回答多表的查询：以下语句可回答“Rawlins 教授的学生中多少人的数学（M200）课得了 A？”的问题：

```
SELECT COUNT (*) FROM vw_students_grades_teachers
WHERE professor = 'Rawlins' AND class = 'M200' AND
grade = 'A'
```

下面语句可回答“学生号为 2110 的学生在各科中都获得了什么分数？谁是其老师？”的问题：

```
SELECT first_name, last_name, class, section, grade, professor
FROM vw_students_grades_teachers WHERE SID = 2110
```

而下面的语句可回答“哪些学生选修了历史系的课程？以及他们的成绩如何？”的问题：

```
SELECT first_name, last_name, class, section, grade, professor
FROM vw_students_grades_teachers
WHERE department = 'History'
```

请注意，在每项查询中，由视图来处理在相关表中的父/子行之间的结合（匹配）问题。因而，多表视图让用户执行简单的（单表的）SELECT 语句来提取原来只能从多个表中才能查询的信息。

技巧 155 使用 UPDATE 语句通过视图改变多个表中的数据

正如读者在技巧 53 “使用 UPDATE 语句改变列值”中所学习的，UPDATE 语句允许改变一个而且只能是一个目标表中的一列或多列中的值。但是以下 UPDATE 语句的句法：

```
UPDATE <table name/view name> SET <column name> =
<expression>[, ..., <last column name> = <last expression>]
WHERE <search condition>
```

似乎表明，在 UPDATE 语句中可以指定基于多个表的视图作为目标“表”。使用视图作为一个且只一个目标表，可在 UPDATE 语句中跟随表名的 SET 子句中引用从不同表中派生出来的视图列。

遗憾的是，DBMS 只允许在 UPDATE 语句中跟随目标表名的 SET 子句中包括来自单个底层（基）表中的列。因此，如果有由以下语句创建的视图：

```
CREATE VIEW vw_customer_invoices AS
SELECT * FROM customers, invoices
WHERE cust_ID = purchased_by
```

而其基表是由以下语句所创建的：

```
CREATE TABLE customers
(cust_ID INTEGER,
name      VARCHAR(30),
address   VARCHAR(50))
```

```
CREATE TABLE invoices
  (invoice_number INTEGER,
   invoice_date    DATETIME,
   purchased_by    INTEGER,
   ship_to         VARCHAR(50))
```

那么以下 UPDATE 语句是可接受的：

```
UPDATE vw_customer_invoices SET address = 'New Address'
  WHERE CUST_ID = 1
UPDATE VW_customer_invoices SET ship_to = 'New Address'
  WHERE CUST_ID = 1
```

因为每条语句更新的只是一个表中的列，即使视图本身是由来自不止一个表中的数据所组成的。但是，DBMS 将不接受以下 UPDATE 语句：

```
UPDATE vw_customer_invoices SET address = 'New Address',
  ship_to = 'New Address'
WHERE CUST_ID = 1
```

因为它试图更新不止一个表中的列。

请记住，DBMS 通过根据所涉及的基表的列生成等价的语句来处理引用视图的 SQL 语句。因而，为了让 DBMS 执行既引用了 ADDRESS（来自 CUSTOMERS 表）又引用了 SHIP_TO（来自 INVOICES 表）列的 UPDATE 语句，DBMS 必须构建一个非法的 UPDATE 语句——即列出两个表而不是一个表作为其目标表的 UPDATE 语句。

注意：MS-SQL Server 提供一种名为“触发器”（trigger）的特殊类型的存储过程，可用来使更新一个表中的一列可引起更新其他表中的列。

技巧 156 在 CREATE VIEW 语句中使用 CHECK OPTION 子句将视图约束应用于 INSERT 和 UPDATE 语句

正如读者在技巧 154 “使用视图显示一个或多个表的特定行中的列”中所学习的，水平视图是只显示来自一个或多个底层表或视图中的某些行的一种虚拟表。由于纵向视图并不显示其基表中的所有行，一个对纵向视图拥有 INSERT 权限的用户可使用视图向视线中不能看到的基表中插入（INSERT）行。类似地，一个拥有 UPDATE 权限的用户也可以这种使行从视线中“消失”的方式改变列值。

例如，假设有由以下语句所创建的纵向视图：

```
CREATE VIEW vw_nv_employees AS
SELECT * FROM employees WHERE office = 'NV'
```

其中有来自以下底层的 EMPLOYEES 表中的数据：

id	name	ssan	office
1	Konrad	555-55-5555	TX
10	Sally	222-22-2222	NV
15	Wally	111-11-1111	NV
28	Walter	333-33-3333	CA

而且执行以下 SELECT 语句：

```
SELECT * FROM vw_nv_employees
```


DBMS 将显示雇员 ID 为 10 和 15 的行中的数据，因为在 EMPLOYEES 表中只有这两行的 OFFICE 列值为 NV。

如果执行以下 INSERT 语句：

```
INSERT INTO vw_nv_employees  
VALUES (2, 'Kris', '777-77-7777', 'TX')
```

DBMS 将向 EMPLOYEES 表中添加新行。但是，如果又执行以下语句：

```
SELECT * FROM vw_nv_employees
```

DBMS 仍然显示雇员 ID 为 10 和 15 的行中的信息，因为插入的新行并不满足视图的搜索条件（OFFICE = 'NV'）。

类似地，如果执行以下 UPDATE 语句：

```
UPDATE vw_nv_employees SET office = 'LA' WHERE id = 10
```

雇员 ID 为 10 的行将从视图中“消失”。虽然 UPDATE 语句并不从底层 EMPLOYEES 表中删除行，但该行不再满足视图的搜索条件，因而也就不再包括在视图中了。

为了防止用户使用视图插入视图并不显示的行，而且防止改变视图列中的数据，从而使行从视图中删除，可向视图定义中添加 WITH CHECK OPTION 子句。在本例中，如果用以下语句创建 VW_NV_EMPLOYEES 视图：

```
CREATE VIEW vw_nv_employees AS  
SELECT * FROM employees WHERE office = 'NV'  
WITH CHECK OPTION
```

DBMS 将只允许用户插入（INSERT）那些 OFFICE 列值为 NV 的行。另外，DBMS 将不允许执行试图将已有行中的 OFFICE 列值改变为不是 NV 的其他值。

例如，虽然以下 UPDATE 语句：

```
UPDATE vw_nv_employees SET office = 'LA' WHERE id = 10
```

当 VW_NV_EMPLOYEES 视图的定义中没有 WITH CHECK OPTION 子句时是完全可接受的，但现在 DBMS 将中止该 UPDATE 语句的执行并返回以下错误消息：

```
Server: Msg 550, Level 16, State 1, Line 1  
The attempted insert or update failed because the target  
view either specifies WITH CHECK OPTION or spans a view  
that specifies WITH CHECK OPTION and one or more row  
resulting from the operation did not qualify under the  
CHECK OPTION constraint.  
The statement has been terminated.
```

技巧 157 在 CREATE VIEW 语句中使用 GROUP BY 子句创建显示总结数据的视图

如果使用没有 GROUP BY 子句的 SELECT 语句创建视图，则在视图的行和被视图显示数据的底层表中的行之间有一对一的对应关系。这样一来，视图中的一个（未成组的）SELECT 语句使得视图就像是一个源表中数据的筛选器，过滤出某些行和列，而让其他行和列通过不发生改变。作为对照，当使用成组的 SELECT 语句（即带 GROUP BY 子句的 SELECT 语句）定义视图时，DBMS 将相关的数据行组合在一起，视图将显示每一组行或来自基表中的一些行的一个结果行。因此，在成组的视图的行（用成组的 SELECT 语句创建的视图）和其基表中

的行没有一对一的对应关系。

创建成组视图的句法如下：

```
CREATE VIEW <view column name list>
AS SELECT <source table column name list>
FROM <source table list> [<WHERE clause>]
GROUP BY <group by column list>
```

这样，为了创建总结顾客定单的成组视图，可使用以下 CREATE VIEW 语句：

```
CREATE VIEW vw_customer_orders
(customer_number, orders_placed_ct, orders_shipped_ct,
total_amt_purchased, total_amt_paid, total_amt_due,
high_order_amt, avg_order_amt, low_order_amt)
AS SELECT
cust_ID, COUNT(*), COUNT(date_shipped),
SUM(invoice_total), SUM(amt_paid), SUM(invoice_total) -
SUM(amt_paid), MAX(invoice_total), AVG(invoice_total),
MIN(invoice_total)
FROM invoices
GROUP BY cust_ID
```

使用成组视图的主要缺点是，它是只读的。由于在视图的行与其源表的行间没有一对一的对应关系，DBMS 不能将 UPDATE 或 DELETE 语句翻译成等价的对视图底层表的特定行执行操作的语句。

另一方面，使用成组视图的主要好处是，它简化了对总结数据的查询。例如，可对成组视图执行以下简单的 SELECT 语句，以便获得有最高到期余额的 10 位顾客的清单：

```
SELECT TOP 10 customer_number, name, total_amt_due
FROM customers, vw_customer_orders
WHERE cust_ID = customer_number
ORDERED BY total_amt_due DESC
```

技巧 158 使用 CREATE VIEW 语句显示组合两个或多个表的结果

除了通过将用户限制为只能访问特定的行和列以提供安全性之外，视图还提供一种将来自许多正规表的相关数据合并为单一表的方法。设计人员将数据库正规化（至少到 3NF），以防止当用户对数据库内容做出改变时可破坏数据库完整性的修改异常。

但是，对正规化数据库即使执行简单的查询，如“列出每个销售人员的顾客”，通常也需要将至少两个表合并，因为顾客数据处于一个表中，而雇员（销售人员）数据将处于另一表中。如果稍微将查询变得复杂一点，添加“……并列出行每个顾客定单的总计和平均量”，编写 SQL 语句的人必须知道如何编写正确地合并 3 个表并执行成组查询的 SELECT 语句。

幸运的是，视图提供了仍然在多个表中存储数据以通过正规化保持数据库的完整性，同时允许用户通过执行简单的单表 SELECT 语句访问数据库信息的方法。换句话说，DBA 和编程人员可编写复杂的将来自多个相关表中的表行合并并将其存储在视图定义中的 SELECT 语句。然后用户就可通过引用视图中的列使用视图来执行涉及几个表的合并的复杂查询，就好像所有相关数据都处于单一表的许多列中一样。

例如，使用以下语句创建的视图：

```

CREATE VIEW vw_cust_invoices
  (cust_id, total_purchased, avg_purchase)
AS SELECT
  customers.cust_id, SUM(invoice_total), AVG(invoice_total)
FROM customers, invoices
WHERE customers.cust_id = invoices.cust_id

CREATE VIEW vw_salesperson_customers
  (salesperson_ID, salesperson_name, customer_ID,
   customer_name, total_purchased, avg_purchase)
AS SELECT
  Employees.ID, employees.name, customers.cust_ID,
  customers.name, total_purchased, avg_purchase
FROM customers, employees, vw_cust_invoices
WHERE customers.salesperson_ID = employees.ID AND
      vw_cust_invoices.cust_ID = customers.cust_id

```

用户可执行以下简单的 SELECT 语句：

```

SELECT * FROM vw_salesperson_customers
ORDER BY salesperson_ID, customer_ID

```

来执行复杂的查询：“列出每个销售人员的顾客以及向每个顾客销售的总量和平均量”。

从本例中学习到的重要内容是，引用合并了来自多个表中数据的视图，允许用户执行单表 SELECT 语句来回答需要将来自几个表中的行数据合并起来才能回答的查询。

技巧 159 使用 UNION 运算符选择出现在任一或全部的两个或多个表中的所有行

UNION 运算符提供了一种组合两个或多个查询结果的方便方法。每当 DBMS 执行 UNION 运算时，最后的结果表包括由 UNION 中的每个查询所产生的所有行，而重复的行都消除了。为了与 UNION 运算兼容，查询必须有以下特性：

- 所有查询必须有同样数量的列。
- 在每个查询结果表中对应列的数据类型必须是同一数据类型（也就是说，如果一个结果表中的第一列是 CHAR 类型，那么所有结果表中的第一列都必须是 CHAR；如果一个结果表中的第二列的数据类型是 INTEGER，那么所有结果表中的第二列都必须是 INTEGER 等）。
- 所有 SELECT 语句都不能有 ORDER BY 子句（在技巧 163“使用 ORDER BY 子句对 UNION 运算的结果排序”中，读者将学习如何对由在最后的 SELECT 语句之后放置 ORDER BY 子句执行 UNION 运算所产生的最后的结果表排序）。

为了使用 UNION 运算符，可将其放在想要合并其结果表的 SELECT 语句之间。例如，假设公司向 3 个卖主转售产品，每个卖主有如下所示的自己的产品表：

```

ABC_Products table
item_no item_desc price  count_on_hand
-----
1      Widget      254.00  5
2      Gidget      123.00  7
3      Gadget      249.00 10

DEF_Products table

```

item_number	description	cost	qty_on_hand
1	Sprocket	243.00	15
2	Gadget	100.00	7

item_number	description	cost	qty_on_hand
7	Laser	575.00	12
10	Phaser	625.00	5
15	Taser	75.00	7

GHI_Products table

item_number	description	cost	qty_on_hand
7	Laser	575.00	12
10	Phaser	625.00	5
15	Taser	75.00	7

可在以下查询中使用 UNION 运算符生成可供货的完整产品清单：

```
SELECT 'ABC' as 'vendor', item_no, item_desc, price,
count_on_hand FROM abc_products
```

UNION

```
SELECT 'DEF' as 'vendor', item_number, description, cost,
qty_on_hand FROM def_products
```

UNION

```
SELECT 'GHI' as 'vendor', item_number, description, cost,
qty_on_hand FROM ghi_products
```

在执行了本例中的查询之后，DBMS 将显示简单的结果表（类似于图 9.1 中所显示的），其中包括由 UNION 运算符合并的每条 SELECT 语句所返回的所有行。

vendor	item_no	item_desc	price	count_on_hand
ABC	1	Gadget	254.0000	5
ABC	2	Gadget	123.0000	7
ABC	3	Gadget	249.0000	10
DEF	1	Sprocket	243.0000	15
DEF	2	Gadget	100.0000	7
GHI	7	Laser	575.0000	12
GHI	10	Phaser	625.0000	5
GHI	15	Taser	75.0000	7

(9 row(s) affected)

图 9.1 显示由 3 条查询的 UNION 运算生成的结果表的 MS-SQL Server Query Analyzer

注意：正如本例中所表明的，UNION 运算符只要求要组合的 SELECT 语句有匹配的对列类型——对应的列名可以是不同的。遗憾的是，某些 DBMS 产品对每个名称不同的对应 SELECT 语句列生成结果表中的无名列（如在本例中 ITEM_NO 与 ITEM_NUMBER 不同，ITEM_DESC 与 DESCRIPTION 不同，PRICE 与 COST 不同以及 COUNT_ON_HAND 与 QTY_ON_HAND 不同）。其他 DBMS 产品，如 MS-SQL Server 将使用 UNION 运算中第一条 SELECT 语句中的列名作为最后结果表中的列标题。

技巧 160 使用 UNION ALL 运算符选择出现在任一或全部的两个或多个表中的所有行（包括重复的行）

正如在技巧 159“使用 UNION 运算符选择出现在任一或全部的两个或多个表中的所有行”中所提到的，UNION 运算符的特点之一是，它能从最后结果表中消除重复的行。UNION 运算符的行为与 SELECT 语句形成了对照，后者（在默认情况下）显示其找到的重复行。由此，为了防止查询返回重复的行，必须向 SELECT 子句中添加 DISTINCT 关键词（如 SELECT DISTINCT * FROM <table name>）。作为对照，由于 UNION 运算符在其结果表中只返回惟一的（不同的）行，如果想要 DBMS 显示所有行（包括重复的行），则必须使用 UNION ALL 来组合查询。

例如，假设有房产存货目录表，由以下语句所创建：

```
CREATE table house_inventory
(address          VARCHAR(50),
sales_price      MONEY,
pool             CHAR(1),
gated_community CHAR(1),
acreage          NUMERIC,
bedrooms         SMALLINT,
square_footage   INTEGER,
realtor_ID       SMALLINT)
```

而想要获得满足购房者的购买理想房屋标准的房产清单。可执行以下 SELECT 语句来获得至少满足购房者期望之一的房产的清单：

```
SELECT * FROM house_inventory
WHERE acreage > 2 OR bedrooms >= 4 OR gated_community = 'Y'
```

但是，如果某些房产只能满足标准之一，某些满足两个，而其他或许满足所有 3 条要求，这时将那些满足最多标准的房产放在清单的前面，而满足最小标准的放在后面可能更为方便。

为了创建以满足购房者标准数排序的房产清单，首先使用与 HOUSE_INVENTORY 表一样的结构创建一个临时表（如 HOUSE_PROSPECTS）。接着使用以下 INSERT 语句，用来自 HOUSE_INVENTORY 表那些满足至少一条购房者期望的行来填充表：

```
INSERT INTO house_prospects
SELECT * FROM house_inventory WHERE acreage > 2
UNION ALL
SELECT * FROM house_inventory WHERE bedrooms >= 4
UNION ALL
SELECT * FROM house_inventory WHERE gated_community = 'Y'
```

UNION ALL 运算符通过组合来自每条 SELECT 语句（中间的）结果表而创建最后的结果表——不过滤重复的行。因此，如果一所房子占地 3 英亩并有 4 个卧室，则 UNION ALL 运算符将把此房产放在最后的结果表中两次（一次是从第一条 SELECT 语句的结果表中来的，一次是从第二条 SELECT 语句的结果表中来的）。INSERT 语句从最后的结果表中提取行并将其插入到 HOUSE_PROSPECTS（临时）表中。

最后，执行以下 SELECT 语句：

```
SELECT COUNT(*) AS 'Score', address, sales_price, acreage,
bedrooms, gated_community FROM house_prospects
```

```
GROUP BY address, acreage, bedrooms, gated_community
        sales_price
ORDER BY score DESC, sales_price ASC
```

DBMS 将显示一个报告，以“得分”的降序排序，得分代表每个房产出现在 HOUSE_PROSPECTS 表中的次数（事实上，得分（或重复的行的计数）告诉人们一所房产满足了几条搜索条件，因为每当 UNION 中的 SELECT 语句之一在中间结果表中包括一行时，UNION ALL 运算符都添加同样的行）。

技巧 161 使用 UNION CORRESPONDING 运算符组合来自两个或多个与 UNION 不兼容的表中的行

如果两个表有同样数量的列而且一个表中每列的数据类型与另一表中的对应列（按顺序位置）的数据类型一样，则两个表是与 UNION 兼容的。例如，由以下语句创建的两个表是与 UNION 兼容的：

```
CREATE TABLE table_a          CREATE TABLE table_b
  (ID          SMALLINT,        (emp_ID        SMALLINT,
  office       INTEGER,         office         INTEGER,
  address      VARCHAR(30),      home_address   VARCHAR(30),
  department   CHAR(5))          emp_department  CHAR(5))
```

因为两个表有同样数量的列而且 TABLE_A 中的第一列与 TABLE_B 的第一列的数据类型相同，TABLE_A 中的第二列与 TABLE_B 的第二列的数据类型相同。

当两个表与 UNION 兼容时，就可在 SELECT 语句间使用 UNION 运算符，如下面的语句：

```
SELECT * FROM table_a
UNION
SELECT * FROM table_b
```

来显示组合了来自 TABLE_A 和 TABLE_B 两个表中的所有行的结果表——同时重复的行已经删除了。如果（或多个）表不是与 UNION 兼容的（既可由于对应列顺序不同，也可由于有的列在另一表中找不到），此时可使用 UNION CORRESPONDING 运算符告诉 DBMS 创建合并两个表中有匹配列名的数据的结果表，而不管每一个表中的顺序位置。

例如，如果有两个由以下语句创建的和 UNION 不兼容的表：

```
CREATE TABLE table_c          CREATE TABLE table_d
  (ID          SMALLINT,        (office       INTEGER,
  office       INTEGER,         ID            SMALLINT,
  address      VARCHAR(30),      address       VARCHAR(30),
  department   CHAR(5),          emp_department CHAR(5))
  pay_rate     MONEY)
```

可与 SELECT 语句一起使用 UNION CORRESPONDING 运算符，如下所示：

```
SELECT * FROM table_c
UNION CORRESPONDING
SELECT * FROM table_b
```

来显示两个表中有共同列名的列中的数据（本例中就是 ID、OFFICE 和 ADDRESS 列）。

注意：并不是所有的 DBMS 产品（包括 MS-SQL Server）都支持 UNION CORRESPONDING 运算符。但不用灰心，正如读者在技巧 162 “使用 UNION 运算符组合两条查询的结果”中将

要学习的, 如果 DBMS 不支持 UNION CORRESPONDING 运算符, 仍然可以通过明确地列出在最后的結果表中要合并的列 (与使用 SELECT * 不同), 从而显示两个与 UNION 不兼容的表的组合的数据。

技巧 162 使用 UNION 运算符组合两条查询的结果

在技巧 159 “使用 UNION 运算符选择出现在任一或全部的两个或多个表中的所有行”中, 读者已经学习了如何使用 UNION 运算符将两个或多个查询结果组合成单一結果表。如果这些表是与 UNION 运算兼容的 (也就是说, 要组合的所有表都有相同数目而且有相同顺序的列), 可使用 UNION 运算符合并来自 “选择所有列” 的查询結果, 例如:

```
SELECT * FROM invoices_99 WHERE invoices_99.cust_ID =  
    (SELECT cust_ID FROM CUSTOMERS WHERE name = 'XYZ Corp')  
UNION  
SELECT * FROM invoices_00 WHERE invoices_00.cust_ID =  
    (SELECT cust_ID FROM CUSTOMERS WHERE name = 'XYZ Corp')
```

以上语句列出由 XYZ CORP 在两年间所下的所有定单。

另一方面, 如果处理的表与 UNION 运算不兼容, 只要明确地列出运算符要合并的列, 仍然可以使用 UNION 运算符来组合查询結果。例如, 假设 Ford Motor Company (福特汽车公司) 宣布了福特探索者的加强型 (多年的所有型号), 而且用户想要获得正等待销售的探索者和已经销售的探索者的完整清单。

假如 AUTO_INVENTORY 和 AUTO_SALES 表有不同的结构, 可以使用 UNION 运算符将列出要合并的列的查询组合成最后的結果表, 如下所示:

```
SELECT store_name AS 'sold_to', address, phone, make,  
    model, vehicle_ID, 'Inventory' AS 'location',  
    date_received  
FROM auto_inventory, dealerships  
WHERE dealerships.store_ID = auto_inventory.dealership_ID  
    AND make = 'Ford' AND model = 'Explorer'  
UNION  
SELECT first_name + ' ' + last_name AS 'sold_to',  
    address, home_phone, make, model, vehicle_ID,  
    'Customer' AS 'location', date_sold  
FROM customers, auto_sales  
WHERE customers.customer_ID = auto_sales.cust_ID  
    AND make = 'FORD' AND model = 'Explorer'
```

虽然每条 SELECT 语句的表有不同的结构, 只要由每条 SELECT 语句返回的列与另一条 SELECT 语句返回的列相匹配 (无论是数量还是数据类型按顺序都匹配), 就可使用 UNION 运算符 (与必须使用 UNION CORRESPONDING 运算符不同) 来合并 (中间的) 結果表。

注意: 某些 DBMS 产品在要由 UNION 运算符组合的查询的 SELECT 子句中只允许列名清单或只允许星号 (*) (其意义为所有列)。而其他产品, 如 MS-SQL Server, 既允许总计函数、简单表达式 (如串接 (FIRST_NAME + ' ' + LAST_NAME)), 也允许实际的字符串 ('Inventory'、'Customer'), 如本例中所示。一定要检查系统手册, 找出具体的 DBMS 产品对由 UNION 运算符连接的 SELECT 语句的特定限制。

技巧 163 使用 ORDER BY 子句对 UNION 运算的结果排序

由 UNION 运算符组合起来的 SELECT 语句不能有 ORDER BY 子句（对 UNION 运算符要合并的中间结果表排序可能是缺乏效率的操作。毕竟，用户看不到中间的结果表）。但是，可把 ORDER BY 子句放在 UNION 中最后的 SELECT 语句之后，以便对最后（合并后的）结果表排序。

例如，为了对类似于技巧 159 “使用 UNION 运算符选择出现在任一或全部的两个或多个表中的所有行”中的查询的最后结果排序，可在最后的 SELECT 语句之后添加 ORDER BY 子句。如下所示：

```
SELECT 'ABC', item_no, item_desc, price, count_on_hand
FROM abc_products
UNION
SELECT 'DEF', item_number, description, cost, qty_on_hand
FROM def_products
UNION
SELECT 'GHI', item_number, description, cost, qty_on_hand
FROM ghi_products
ORDER BY item_no
```

ORDER BY 子句可指定按最后的结果表中的任何列名排序。在本例中，DBMS 将以 ITEM_NO 列值的升序对最后的结果表排序，因为 MS-SQL Server（方便地）将最后结果表中的第 3 列命名为 ITEM_DESC。遗憾的是，某些 DBMS 产品并不对组合了对应表列使用不同列名的查询的表列命名。如果正在使用这类产品之一，系统将让本例中的最后结果中的所有列都为未命名状态，因为第一列是实际的字符串并没有列名，而且还因为其余列在查询的 3 个表中有不同的名称。

为了使用 ORDER BY 子句对最后结果表的内容按未命名的列排序，可引用结果表中列的顺序位置。例如，为了对本例中的结果表按 ITEM_NO 排序，可将 ORDER BY 子句写成：

```
ORDER BY 2
```

因为 ITEM_NO 是最后结果表中的第二列。类似地，为了在本例中按实际字符串（正好是销售商 ID）和货品描述对最后结果表排序，可以使用 ORDER BY 子句。）

```
ORDER BY 1, 3
```

因为实际的字符串（卖主 ID）是结果表中的第一列而货品描述是结果表中的第 3 列。

技巧 164 使用 UNION 运算符组合 3 个或 3 个以上的表

正如读者在技巧 159~技巧 162 中所学习的，UNION 运算符可用于将来自多个 SELECT 语句的结果组合成单一的复合结果表。不管是使用单个 UNION 运算符组合两条 SELECT 语句的输出还是使用 10 个 UNION 运算符组合来自 11 条 SELECT 语句的结果，查询的合并总是一次执行两个结果表。

例如，假设要执行以下语句：

```
SELECT * FROM table_a UNION
SELECT * FROM table_b UNION
SELECT * FROM table_c
```

DBMS 将根据如图 9.2 所示的工作流程来执行。

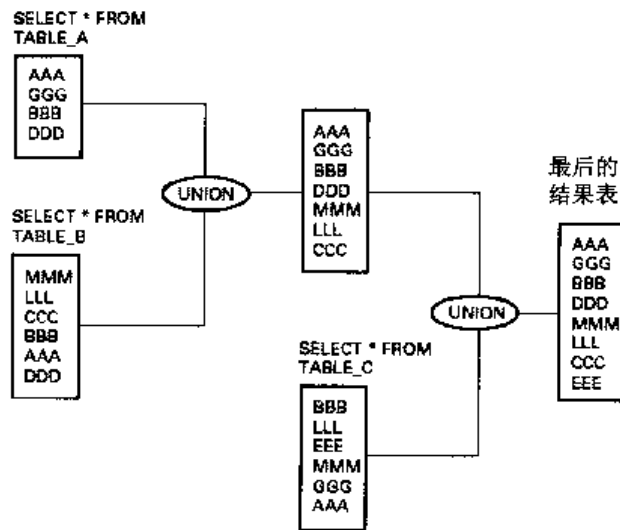


图 9.2 3 个查询结果表的合并 (UNION)

可以使用括号()告诉 DBMS 组合 SELECT 语句结果表的顺序。但是, 如果所有的 SELECT 语句都要使用 UNION 运算符组合, 或者如果所有的 SELECT 语句都要使用 UNION ALL 运算符组合, 最后的结果表总是一样的, 而不管合并时的顺序如何。

当把 UNION 和 UNION ALL 运算符混合使用时, 执行合并时的顺序变得重要。例如, 假如表数据如图 9.2 所示。

```

table_a UNION (table_b UNION table_c)
(table_a UNION table_b) UNION table_c
(table_a UNION table_c) UNION table_b
  
```

都会产生同样的结果表, 其中有 8 个不重复的行。类似地, 按以下组合执行 UNION ALL 语句:

```

table_a UNION ALL (table_b UNION ALL table_c)
(table_a UNION ALL table_b) UNION ALL table_c
(table_a UNION ALL table_c) UNION ALL table_b
  
```

将产生同样的有 16 行的最后结果表 (请记住, UNION ALL 运算符并不清除重复的行)。

但是, 如果想要让 DBMS 执行:

```
table_a UNION ALL table_b UNION table_c
```

如果系统执行顺序如下, 则结果是不同的:

```
table_a UNION ALL (table_b UNION table_c)
```

这将生成有 13 行 (从内部的 table_b 和 table_c 合并 (UNION) 有 9 个不重复的行, 再加上外部的与 table_a 的 UNION ALL 产生的 4 行), 这与下面的运算不同:

```
(table_a UNION ALL table_b) UNION table_c
```

此运算只产生 8 行的结果 (按本例中的数据, 在内部的 table_a 和 table_b 的 UNION ALL 产生 10 行, 而外部的 UNION 添加行 EEE, 但消除了重复的行 AAA 和 DDD)。

要理解的重要事情是, 如果只使用 UNION 或只使用 UNION ALL 运算符来组合查询结果, 不必告诉 DBMS 执行合并的顺序。但是, 如果混合使用 UNION 和 UNION ALL 运算符, 则应该总是使用括号指定执行的顺序, 以便系统只抛弃那些重复的想要清除的行。

技巧 165 理解 MS-SQL Server 的事务处理日志放于何处才能改善性能

DBMS 使用事务处理日志来做以下事情：

- 当用户执行 ROLLBACK 语句时取消完成的未提交工作。
- 如果恢复数据库的备份版本并应用较新的事务处理日志，重做由完成的事务处理执行的工作。
- 在系统断电后 DBMS 重新启动时，重做提交但未写入数据库文件的工作。
- 在系统断电后 DBMS 重新启动时，取消未提交的事务处理。

为了将事务处理日志用于这些目的，DBMS 必须用其保存对数据库所执行的所有工作的顺序清单，还要保存数据库中的每个修改值的修改前后的值。事实上，大多数 DBMS 产品，在改变数据库表本身之前将原始的和更新后的数据值写入事务处理日志。因此，事务处理日志的物理位置对数据库的总体性能有重要的影响。

只要可能，应将事务处理日志放在快速的物理上分开的磁盘驱动器上或是 RAID 阵列上。由于 DBMS 要通过把新信息追加到日志文件尾部而维护事务处理日志，将日志放在专门的驱动器上将使磁头留在原处以便下一次的写操作。结果，DBMS 就不必在更新之前等待系统硬件来重新寻找日志的尾部。虽然几个毫秒似乎不是太多的时间，但当考虑到每条命令和每个修改值的两个副本以及处理其他的 SQL 语句都加在一起时，时间就会越积越多。

技巧 24“使用 CREATE DATABASE 语句创建 MS-SQL Server 数据库和事务处理日志”已经向读者展示了如何作为 CREATE DATABASE 语句的一部分为事务处理日志指定位置（文件名和物理磁盘存储设备），而技巧 25“使用 MS-SQL Server Enterprise Manager 创建数据库和事务处理日志”解释了如何使用 MS-SQL Server Enterprise Manager 来做同样的事情。请参考那些技巧中的创建事务处理日志的特定步骤。

现在，要理解的重要事情是，应该在分开的（专门的）驱动器上创建大型的事务处理日志。这样做将改善 DBMS 的性能，因为系统不必扩展日志的容量或是在记录命令和数据值以更新数据库并继续执行下一条语句之前等待硬件来寻找文件的尾部。

技巧 166 理解多列的 UNIQUE 约束

在技巧 144“使用 UNIQUE 列约束防止列中的重复值”中，读者学习了如何对列定义应用 UNIQUE 列约束以便保证 DBMS 防止在列中存储重复的数据值。当创建下面的表时，单列的 UNIQUE 约束是合适的：

```
CREATE TABLE employees
(
    employee_ID SMALLINT,
    office       SMALLINT,
    emp_name     VARCHAR(30),
    SSAN        CHAR(11) UNIQUE)

```

其中 SSAN（社会安全账号）列中的值必须对表中的每行是惟一的，因为没有两个雇员会有相同的社会安全账号。但是，假如公司有多个办公室而且每个办公室都有其自己的雇员号序列。因此，虽然 OFFICE 为 1 的办公室只能有一个人的雇员 ID 号为 101，但 OFFICE 2 也可有一个（且只能有一个）人的雇员 ID 号为 101。类似情况，对 OFFICE 3 等也是一样的。

当将两个或多个列值组合必须惟一时，就不能使用多个单列 UNIQUE 约束，例如：

```
CREATE TABLE employees
  (employee_ID SMALLINT UNIQUE,
   office        SMALLINT UNIQUE,
   emp_name      VARCHAR(30),
   SSAN          CHAR(11) UNIQUE)
```

如果向每个列定义中添加 UNIQUE 约束，如在本例中所示的那样，DBMS 将对每个办公室只允许插入一个雇员行，因为一个办公室号码，例如 1，只能出现在表中的一行的 OFFICE 列中。作为对照，如果仅对 EMPLOYEE_ID 列应用 UNIQUE 约束，那么 DBMS 就不允许对不同的办公室重复 EMPLOYEE_ID 值。

为了应用多列的 UNIQUE 约束（其中来自两列或多列的一套值必须在整个表中是惟一的），可在列定义之外定义约束。例如，在本例中为了约束 EMPLOYEES 表，使得雇员号必须在每一办公室内是惟一的，但不同的办公室可以重复，可使用以下的 CREATE TABLE 语句：

```
CREATE TABLE employees
  (employee_ID SMALLINT,
   office        SMALLINT,
   emp_name      VARCHAR(30),
   SSAN          CHAR(11) UNIQUE,
   CONSTRAINT unique_by_office UNIQUE (employee_ID, office))
```

UNIQUE_BY_OFFICE 约束将允许插入(EMPLOYEE_ID,OFFICE)对的值为(101,1)、(101,2)、(102,2)、(102,3)的行，但不允许插入多于一行 OFFICE 值为 1 且 EMPLOYEE_ID 为 101 的行，即不允许再插入列对的值为(101,1)的行。

第 10 章 使用函数、参数和数据类型

技巧 167 理解实际值

实际值是用来将值放入表列的常数，既可用在以下语句中：

```
INSERT INTO employees (emp_ID, name, hire_date, hourly_rate)
VALUES (1, 'Konrad King', '08/20/2000', 52.75)
```

（此语句告诉 DBMS 使用列在 VALUES 子句中的实际值向 EMPLOYEES 表中插入一行）也可用在以下语句的表达式中：

```
UPDATE employees SET hourly_rate = 10.0 + hourly_rate * 1.2
```

（此语句告诉 DBMS 将 HOURLY_RATE 列的值乘以数值实际值 1.2，再在积上加上数值实际值 10.0，并将新值存储回 HOURLY_RATE 列）。

每种 DBMS 产品对每一种所支持的标准 SQL 数据类型都有实际值类型，可用在表达式和列值赋值上。表 10.1 显示的是 MS-SQL Server 支持的某些数据类型的实际值的例子。

表 10.1 MS-SQL Server 数据类型和实际值的示例

数据类型	实际值示例
BINARY、VARBINARY、UNIQUEIDENTIFIER	0xab, 0x451245, 0x2586AB
CHAR、VARCHAR、NCHAR、NTEXT、NVARCHAR、TEXT、SYSNAME	'Konrad King', 'Konrad's Tips', 'Hello'
DATETIME、SMALLDATETIME	'April 15, 2000 05:25pm', '08/20/2000'
DECIMAL、MONEY、SMALLMONEY	458.878, 789.5698, -58.78, -785.12
FLOAT、NUMERIC、REAL	258.5E10, -258.45E5, -237E-5, 78.59
INT、INTEGER、SMALLINT	-258, 32767, -15789, -987, 478
TINYINT	1, 0, 255, 200, 75

注意：单引号用来表示非数值的实际值，如字符串和日期。为了使用本身包括单引号的字符串，可在一行上写两个单引号。例如，以下 INSERT 语句：

```
INSERT INTO books (author, title)
VALUES ('Konrad King', 'Konrad's Tips')
```

向 BOOKS 表中添加一行，其中 AUTHOR 列的值为 Konrad King，而 TITLE 列的值为 Konrad's Tips。

技巧 168 使用 SUBSTRING 函数提取部分字符串

当必须从二进制的字符串、表列中的字符串值或是字符串表达式中提取部分字符时可使用 SUBSTRING 函数。例如，为了从一个字符串中提取第 7 位到第 12 位的字符，可使用如下 SUBSTRING 函数的句法：

SUBSTRING(<target string>, <start>, <length>)

可将此句法用在 SELECT 语句中, 如:

```
SELECT SUBSTRING ('King, Konrad', 7, 6)
```

此语句将显示来自实际的字符串"King, Konrad"中的从第 7 个字符开始的 6 个字符 (即位置为 7~12 的字符):

```
-----
Konrad
```

虽然提取部分字符串实际值 (如本例中的 King, Konrad) 在本例中是简单的, 但当将其用于提取表列中的部分字符串并将结果字符串用作 WHERE 子句中的条件时, SUBSTRING 函数的实际能力才真正显示出来。例如, 假设想要列出姓中以 KIN 开头的所有雇员的清单, 可在 SELECT 语句的 WHERE 子句中使用 SUBSTRING 函数, 如下所示:

```
SELECT last_name, SUBSTRING(last_name,1,3) AS 'First_Three',
       SUBSTRING(last_name,4,15) AS 'Remainder'
FROM employees WHERE SUBSTRING(last_name,1,3) = 'KIN'
```

如果不知道字符串的实际长度, 只要指定让 SUBSTRING 函数返回的最大数目的字符串。当 SUBSTRING 函数发现<target string>从<start> (起点) 到字符串尾部的长度比<length>个字符短时, 就返回从<start>位置到结尾时的字符串 (不报告错误)。由此, 如果 EMPLOYEES 表中有 3 个雇员的姓为 KING、KINGSLY 和 KINGSTON, 在本例中的 SELECT 语句将产生以下结果表:

Last_name	First Three	Remainder
King	Kin	g
Kingsly	Kin	gsly
Kingston	Kin	gston

即使最长的姓也只有 8 个字符, 而 SUBSTRING 函数却指定返回第 4 个~第 18 个字符。

注意: 如果<target string>是 NULL, SUBSTRING 函数将返回 NULL 值。类似地, 如果指定的<start>比<target string>的长度还大, 某些 DBMS 产品将返回 NULL 结果。另一些, 例如 MS-SQL Server 将返回空的字符串 (长度为 0 的字符串)。

技巧 169 使用 DISTINCT 子句消除行集中的重复

未受 PRIMARY KEY 或 UNIQUE 约束的单独列可在表的两行或多行上有重复的值。通过向查询中添加 DISTINCT 语句, 可告诉 DBMS 从 SELECT 语句的结果表中消除重复的行。例如, 假设想要了解在过去的 90 天里有多少顾客下了定单。执行以下查询:

```
SELECT COUNT(customer_ID) FROM orders
WHERE order_date >= GETDATE() - 90
```

如果任何一个顾客在此期间下了不止一个定单, 则上面语句将显示不正确的顾客计数。但是, 如果向同一查询中添加 DISTINCT 子句:

```
SELECT COUNT(DISTINCT customer_ID) FROM orders
WHERE order_date >= GETDATE() - 90
```

DBMS 将从结果表中清除重复的顾客 ID 号并给出在此期间下了定单的惟一顾客 ID 的准确计数 (因而也就是顾客数)。

除了 COUNT(<column name>)之外, 还可在 SUM、AVG 和 COUNT(*)等总计函数中使用

DISTINCT 子句。例如，假如 ITEM_COUNT 值为：

```

item_count
-----
4
5
6
4
4
5

```

执行图 10.1 中上窗格中的查询将产生显示在下窗格中的结果表。

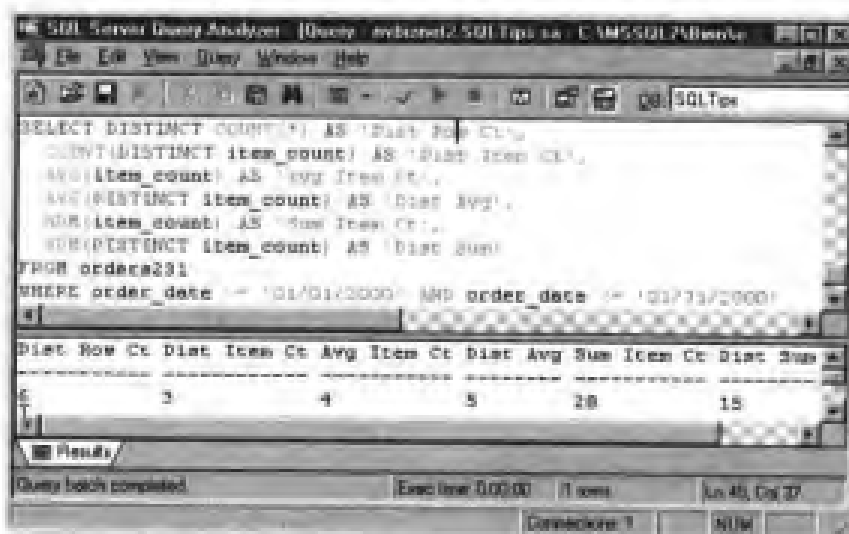


图 10.1 在上（输入）窗格查询中使用 DISTINCT 子句而在下（输出）窗格中显示结果表的 MS-SQL Server Query Analyzer 窗口

请注意，DISTINCT COUNT(*)函数初看起来好像产生了不正确的结果。毕竟，只有 3 个惟一的 ITEM_COUNT 值（4, 5, 6），而 DISTINCT COUNT(*)函数返回的值却为 6。

不一致的原因是，当在列名前面放 DISTINCT 子句时，DBMS 将只清除那些在组合的列中有重复值的行。例如，执行以下查询：

```

SELECT DISTINCT customer_ID, item_count FROM orders
WHERE order_date >= '01/01/2000'
AND order_date <= '01/31/2000'

```

可产生如下的结果表：

```

customer_ID item_count
-----
101         4
102         4
102         5
102         6
103         5

```

虽然两个输出列中的每一个（CUSTOMER_ID 和 ITEM_COUNT）中都有重复的值，但对两列的组合（CUSTOMER_ID 和 ITEM_COUNT）来说结果表却没有重复值。类似地，在前面的例子中，DBMS 将 COUNT(*)展开为“所有列”。因此，DISTINCT COUNT(*)中的 DISTINCT

子句将只清除那些在一行上的所有列中值与另一行完全重复的行。

技巧 170 使用 Transact-SQL 的 STUFF 函数将字符串插入另一字符串

MS-SQL Server 允许使用 STUFF 函数将一个字符串插入另一字符串。STUFF 函数调用的句法如下：

```
STUFF(<string 1>, <starting position>,  
      <length to delete from string 1>, <string 2>)
```

由此，以下 PRINT 语句：

```
PRINT STUFF('**Konrad King's Tips**',16,0,' 1001 SQL')
```

在 STUFF 函数将<character string 2>（即字符串' 1001 SQL'）插入<string 1>（'**Konrad King's Tips**'）的第 16 个字符后面之后，将显示以下结果：

```
**Konrad King's 1001 SQL Tips**
```

如果想让 MS-SQL Server 在插入<string 2>之前删除部分<string 1>，可把<length to delete from string 1>设置为大于 0 的数。例如，以下 PRINT 语句：

```
PRINT STUFF('**Konrad King's Tips**',9,7,' 's 1001 SQL')
```

在 STUFF 函数从<string 1>中第 9 个字符（即**Konrad 之后的空格）开始删除 7 个字符，然后将<string 2>插入<string 1>的同一位置之后（7），将显示以下结果：

```
**Konrad's 1001 SQL Tips**
```

注意：如果<starting position>或<length to delete from string 1>是负的，或者如果<starting position>是大于<string 1>长度的数，则 STUFF 函数将返回 NULL 值。

可使用变量和数据类型为 CHARACTER（或 VARCHAR）的列作为 STUFF 函数中的参数。例如，为了显示 FIRST_NAME 的第一个字母后跟 LAST_NAME，则可使用以下 SELECT 语句：

```
SELECT STUFF(first_name,2,40,'. ' + last_name)  
AS 'First Initial & Last Name' FROM employees
```

来产生以下输出：

```
First Initial & Last Name  
-----  
K. KING  
S. FIELDS  
D. JAMSA  
(3 row(s) affected)
```

注意：如果指定的<length to delete from string 1>长于从<starting position>位置到<string 1>结尾的字符数，STUFF 函数并不报告错误或警告。此函数不过是将<string 1>在<starting position-1>处截短，如在本例中所表明的。

技巧 171 使用 Transact-SQL 的串接运算符“+”在另一字符串尾部添加字符串

MS-SQL Server 允许使用加号（+）运算符将一个字符或二进制字符串表达式的结果追加到另一个字符或二进制字符串表达式上，其句法如下：

```
<string expression> + <string expression>  
[...+ <last string expression>]
```

例如，以下 PRINT 语句：

```
SELECT 'He' + 'llo' + ' ' + world!'
```

将显示以下结果表：

```
-----
Hello world!
```

像通常一样，串接运算符实际价值是其串接字符串实际值、其他字符串函数结果以及 CHARACTER（和 VARCHAR）数据类型的列值的能力。例如，以下 SELECT 语句：

```
SELECT first_name + ' ' + last_name + ' wrote: "' + title +
       '" for ' + publisher + ' circa ' +
       CONVERT(VARCHAR(11),publish_date) + '.'
FROM authors, titles
WHERE author_ID = author
ORDER BY last_name, first_name
```

可用于组合来自 AUTHORS 和 TITLES 表中的实际字符串和列值以形式英语句子，如在以下结果中所示的：

```
-----
Kris Jamsa wrote: "Java Programmer's Library" for Jamsa
Press circa Jun 1 1996.
Kris Jamsa wrote: "1001 Windows 98 Tips" for Jamsa Press
circa Jun 1 1998.
Konrad King wrote: "SQL Tips & Techniques" for Prima
Publishing circa Feb 1 2002.
Konrad King wrote: "Hands on PowerPoint 2000" for Jamsa
Press circa Aug 1 1999.
```

技巧 172 使用 INTERSECT 运算符选择出现在所有两个或多个源表中的行

INTERSECT 运算符生成只包括那些在由该运算符连接的两个数据库表（或查询结果表）中都有的行的结果表，即一个表中与另一表重复的行组成的结果表。例如，执行以下语句将生成包括来自表 A 且在表 B 中也有的行的结果表：

```
(SELECT * FROM a) INTERSECT (SELECT * FROM b)
```

如两个表是与 UNION 运算兼容的，可使用 INTERSECT（或 UNION 或 EXCEPT）运算符来组合两个在 SELECT 子句中使用星号（*）指定“所有列”的两条 SELECT 语句（正如读者在技巧 159“使用 UNION 运算符选择出现在任一或全部的两个或多个表中的所有行”中所学习的，如果两个表有同样数据类型的同样数目的列，且对应的列有相同的顺序，则这两个表与 UNION 运算兼容）。由此由以下语句所创建的两个表就不是与 UNION 运算兼容的：

```
CREATE TABLE a          CREATE TABLE b
  (make, VARCHAR(15),      (model VARCHAR(15),
  model, VARCHAR(15),      make VARCHAR(15),
  sales_price MONEY)       sales_price MONEY)
```

因为对应列（表 A 中的 MAKE 与表 B 中的 MAKE 以及表 A 中的 MODEL 与表 B 中的 MODEL）在两个表中的顺序不同。

只要明确地指定让运算符匹配的来自每个表中的列，仍然可以使用 INTERSECT 运算符组合查询结果表。例如，给定在前面段落中描述的两个表，通过执行以下语句，可使用 INTERSECT 运算符组合对每个表的查询结果：


```
(SELECT make, model FROM a)
INTERSECT
(SELECT make, model FROM b)
```

注意：当明确地指定要组合的列时，INTERSECT 运算符只在列出的列组合中查找重复的。由此，在本例中，结果表将包括那些 (MAKE, MODEL) 组合在两个表为相同的行——即使表 A 中没有与表 B 中相匹配的 SALES_PRICE 列也是如此。

事实上，INTERSECT 运算符使用 AND 运算符而不是 UNION 运算符（此运算符使用的是 OR 运算符）来组合两项查询。例如，如果想要了解“什么产品卖主 A 提供或者卖主 B 提供或者卖主 C 提供？”，可使用 UNION 运算符来连接 SELECT 语句，如下所示：

```
(SELECT vendor_ID, product, price FROM product_list_A)
UNION
(SELECT vendor_ID, product, price FROM product_list_B)
UNION
(SELECT vendor_ID, product, price FROM product_list_C)
```

另一方面，如果想要了解“什么产品卖主 A 提供卖主 B 也提供卖主 C 也提供？”，此时可使用 INTERSECT 运算符来连接两条查询，如下所示：

```
(SELECT vendor_ID, product, price FROM product_list_A)
INTERSECT
(SELECT vendor_ID, product, price FROM product_list_B)
INTERSECT
(SELECT vendor_ID, product, price FROM product_list_C)
```

技巧 173 使用 EXCEPT 运算符选择出现在一个表而不出现在另一表中的行

EXCEPT 运算符是三个数学集合运算符（UNION、INTERSECT 和 EXCEPT）之一，用来组合来自两个或多个查询的结果表。虽然 a INTERSECT b（此运算符已经在技巧 172 “使用 INTERSECT 运算符选择出现在所有两个或多个源表中的行”中学过）生成的结果表中的行是来自表 A 且在表 B 中重复的，但表达式 a EXCEPT b 生成结果表中的行是来自表 A 且在表 B 中不重复的。

对于诸如下面的一类的查询，EXCEPT 运算符是有用的。

“请给出仅由卖主 A 提供的产品清单”：

```
((SELECT description FROM vendor_a_products) EXCEPT
(SELECT description FROM vendor_b_products)) EXCEPT
(SELECT description FROM vendor_c_products)
```

“给出还没有购买 Widget 的顾客的清单”：

```
(SELECT customer_ID FROM customers) EXCEPT
(SELECT (customer_ID FROM orders WHERE item = 'Widget'))
```

“给出还没有选 Chemistry 101 课的新生的清单”：

```
(SELECT student_ID FROM students
WHERE class = 'freshman') EXCEPT
(SELECT student_ID FROM classes WHERE class_ID = 'CHEM 101')
```

如果两个表是与 UNION 运算符兼容的（也就是说，两个表有相同数目的列，对应的列有相同的数据类型且以相同的顺序出现），可在查询的 SELECT 子句中使用“所有列”运

算符(*)来生成包括来自第一个表中且在第二个表中不重复的行的结果表。例如,如果有两个销售办公室,而且两个销售办公室的存货表 INVENTORY_O1(1号办公室用)和 INVENTORY_O2(2号办公室用)有相同的结构,为了获得只在1号办公室才有的存货货品的清单,可执行以下语句:

```
(SELECT * FROM inventory_o1) EXCEPT
(SELECT * FROM inventory_o2)
```

注意:正如在本技巧开始的3个示例所表明的那样,可使用 EXCEPT 运算符组合对不兼容于 UNION 运算的表的查询,只要在每条查询的 SELECT 子句中列出对应的列,而不是使用所有列运算符(*)。

技巧 174 使用 POSITION 函数返回字母或子字符串在字符串中的位置

POSITION 函数允许在源字符串中搜索目标字符串并返回目标字符串的开头第一次出现在源字符串中的位置 (INTEGER 类型)。例如,以下语句:

```
PRINT POSITION ('k' IN 'My name is Konrad')
```

将显示 INTEGER 类型的值 12, 因为根据 POSITION 函数调用的句法:

```
POSITION (<target string> IN <source string>)
```

DBMS 将搜索<source string>('My name is Konrad')中第一次出现<target string>(k)的位置,并返回<target string>在<source string>开始处的字符位置(12)。

注意: POSITION 函数执行的是大小写敏感的还是大小写不敏感的搜索要依赖于 SQL 服务器的排序设置。如果服务器的排序设置是大小写不敏感的,那么大写字母与小写字母有相同的值。因此,对于比较测试和子串搜索来说, K = k。本例假定服务器的排序设置是大小写不敏感的。如果排序设置是大小写敏感的,以下语句将显示整数值 0:

```
PRINT POSITION ('k' IN 'My name is Konrad')
```

因为对于大小写敏感的系统来说, K 与 k 是不相等的,而在<source string>('My name is Konrad')中没有 k。

如果<target string>是由不止一个字符组成的, POSITION 函数搜索<source string>找出第一次字母顺序匹配。例如,以下语句:

```
PRINT
POSITION ('Kingsly' IN 'My last name is king, not kingsly')
```

将显示整数值 27, 因为<target string>'Kingsly'开始在<source string>的第 27 个字母处。

注意: SQL-92 规范要求有 POSITION 函数。但是,具体的 DBMS 产品可能有该函数的不同名称(和句法)。例如,在 MS-SQL Server 上, POSITION 函数是以 CHARINDEX 及以下句法实现的:

```
CHARINDEX(<target_string>,
<source string>[, <start location>])
```

其中指定 CHARINDEX 函数在<source string>字符串从<start location>位置处开始搜索。由此,以下 PRINT 语句:

```
PRINT
CHARINDEX('King','My last name is king, not kingsly', 18)
```

将显示整数 27 而不是 17 (king 在<source string>中的第一次出现的开始位置), 因为 CHARINDEX 在字符位置 18 处开始搜索,因而找到了 kingsly 里的 king 是<target string>在部

分<source string>中的第一次出现。

正如其他字符串函数一样，POSITION（在 MS-SQL Server 上是 CHARINDEX）的真正功能是其接受任何 CHARACTER（或 VARCHAR）数据类型对象或表达式作为其<source string>或<target string>的能力。例如，在以下 SELECT 语句中的 CHARINDEX 函数：

```
SELECT employee_name AS 'Full Name',
       SUBSTRING(LTRIM(employee_name),1,
       CHARINDEX(' ',LTRIM(employee_name))-1) AS 'First Name',
       SUBSTRING(LTRIM(employee_name),
       CHARINDEX(' ',LTRIM(employee_name))+1,30)
       AS 'Last Name'
FROM employees
```

使用（对表列执行的）LTRIM 函数返回的结果，而 SUBSTRING 函数在表达式中使用了 CHARINDEX 函数的结果从 EMPLOYEE_NAME 列中提取字符串，然后 SELECT 语句使用提取的字符串生成以下结果表：

Full Name	First Name	Last Name
Konrad King	Konrad	King
Sally Fields	Sally	Fields

技巧 175 使用 CHAR_LENGTH 函数返回字符串变量的长度

CHAR_LENGTH 函数返回字符串中的字符数。例如，以下语句：

```
PRINT CHAR_LENGTH('Captain Kirk')
```

将显示整数值 12。虽然 SQL 标准指定的 CHAR_LENGTH 函数的句法为：

```
CHAR_LENGTH (<string expression>)
```

但具体的 DBMS 产品可能会使用 CHARACTER_LENGTH、LENGTH 或 LEN 为其函数名。例如，MS-SQL Server 使用 LEN 来返回字符串表达式中排除了尾部空格后的字符数。

与其他字符串函数的情况一样，为了说明性的目的使用字符串实际值是有用的。但是，CHAR_LENGTH 函数的真正价值在于其返回包括列引用在内的字符串表达式的长度的能力。例如，在 MS-SQL Server 上，以下语句：

```
SELECT MAX (LEN(CONVERT(VARCHAR,qty))) AS 'MAX Digits Qty'
       MAX (LEN(description + units)) AS 'MAX Len Desc + Units'
       MAX (LEN(CONVERT(VARCHAR,unit_cost)))AS 'MAX Len Unit Cost'
```

将生成下面的结果表：

MAX Digits Qty	MAX Len Desc + Units	MAX Len Unit Cost
2	31	5

接着就可将结果用于将输出（如图 10.2 所示）格式化为更易阅读的格式（如图 10.3 所示）。

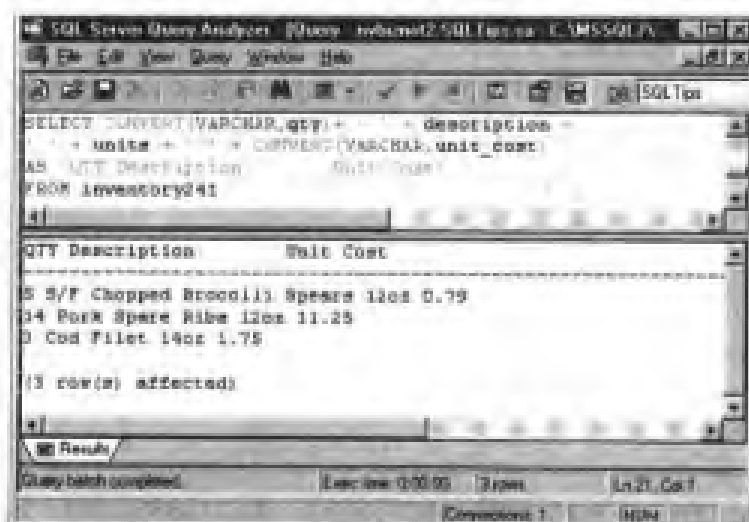


图 10.2 组合了 INTEGER、MONEY 和可变字符串数据的 SELECT 语句

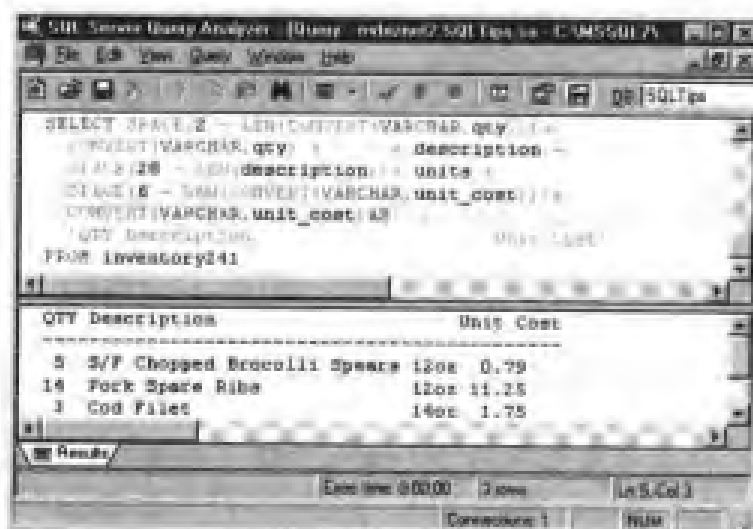


图 10.3 使用 LEN 函数将可变长度的字符串转化为固定长度的字符串的 SELECT 语句

技巧 176 使用 OCTET_LENGTH 函数决定用于保存字符串变量或实际值所需的字节数

计算机使用位 (bits) —— 单独的 1 和 0 —— 在内存中或磁盘上存储数据。保存一个字符 (如一个字母、符号或数字) 所需的最小位数叫做一个字节 (byte)。过去, 许多计算机系统使用 8 位来代表一个字符。结果, 8 位 (一段时间以来) 就是指的一个字节的数据。遗憾的是, 术语“字节”并未指明位数。为了弥补这一不足, 在对函数 OCTET_LENGTH 命名时, SQL 的设计者决定借助于音乐术语 octet (意义为 8 个音节的组合)。OCTET_LENGTH 函数返回存储传递到函数中的参数的值所需的 8 位字节的数目。

例如, 在 8 位字节系统上, 以下语句:

```
PRINT OCTET_LENGTH ('Yellow Brick Road')
```

将显示整数 17, 因为系统将使用 8 个字节来表示每个字符。类似地, 以下语句:

```
PRINT OCTET_LENGTH (0x1001000110011101)
```

将显示整数 2，因为存储 268,505,345 的二进制值占用了两个 8 位字节。

注意：并不是所有的 DBMS 产品都支持 OCTET_LENGTH 函数。例如，MS-SQL Server 将 OCTET_LENGTH 定义为保留字 (Reserved Word)，但没有函数可返回存储一个值或映象所需的字符数 (LEN 函数返回存储一个值所需的字符数，而不是字节[或 octet]数)。要记住的重要事情是，如果 DBMS 支持 OCTET_FUNCTION，系统通过存储一个值或映象所需的位数除以 8 并舍入到最近整数，而计算出结果。

技巧 177 使用 BIT_LENGTH 函数决定用于保存字符串变量或实际值所需的位数

虽然 OCTET_LENGTH 函数返回保存数值或映象所需的 8 位字节数，但 BIT_LENGTH 函数返回所需的位 (1 和 0) 数。因此，以下语句：

```
PRINT BIT_LENGTH (0x100111011111)
```

将显示整数值 12，因为二进制值包括了 12 个 1 和 0。类似地，以下语句：

```
PRINT BIT_LENGTH ('Yellow Brick Road')
```

在那些使用 8 位来保存一个字符的系统上将显示整数值 136。

注意：与 OCTET_FUNCTION 的情况一样，许多 DBMS 产品不支持 BIT_LENGTH 函数。例如，MS-SQL Server 将 BIT_LENGTH 定义为保留字，但没有同样作用的函数。因而，请检查系统手册，在将 BIT_LENGTH 函数用于代码之前，确保其是可用的。大多数 DBMS 产品确实支持返回对象中字符数目的 LEN (CHAR_LENGTH 或 CHARACTER_LENGTH) 函数。由此，如果具体的 DBMS 不支持 BIT_LENGTH 函数，可使用 LEN 函数来计算对象 (如一系列中存储的数据类型为 IMAGE 的图片) 中的位数，只要将对象的字符长度 (LEN 函数返回的) 乘以具体的系统上每字节的位数即可。

技巧 178 使用 EXTRACT 函数从 DATETIME 值中提取单个域

虽然 DBMS 通常将日期和时间存储为诸如 2000-09-04 17:03:35.640 这样的单个值，但有时只需要其中的一个域，如月、日或年。SQL 的 EXTRACT 函数提供一种将 DATETIME 值提取为或分离为其组成部分的方法。例如，以下语句将显示值 4：

```
PRINT EXTRACT(DAY FROM '2000-09-04 17:03:35.640')
```

类似地，以下语句将显示值 3：

```
PRINT EXTRACT(MINUTE FROM '2000-09-04 17:03:35.640')
```

遗憾的是，某些 DBMS 产品不支持 EXTRACT 函数。对于那些支持的产品，EXTRACT 函数语句的句法如下：

```
EXTRACT(<datetime field> FROM <datetime value>)
```

其中：

<datetime field>

也可是

YEAR, MONTH, DAY, HOUR, MINUTE,

SECOND, TIMEZONE_HOUR

或

TIMEZONE_MINUTE

<datetime value>是列、实际值或带有 DATETIME 类型值的表达式。

如果具体的 DBMS 产品不支持 EXTRACT 函数，可能必须使用 CONVERT 函数或 CAST 函数（读者将在技巧 183 “使用 CAST 函数将值从一种数据类型转化为另一种”中学习）将 DATETIME 值转化为 CHARACTER 字符串。在将 DATETIME 值转化为 CHARACTER 字符串之后，就可使用 SUBSTRING 函数（读者已在技巧 168 “使用 SUBSTRING 函数提取部分字符串”中学习过）提取 DATETIME 值中想要的部分。

例如，在 MS-SQL Server 上执行以下语句：

```
DECLARE @date_string VARCHAR(19), @month INTEGER
SELECT @date_string = CONVERT(VARCHAR, GETDATE(), 120)
SELECT @month = SUBSTRING(@date_string, 5, 2)
PRINT @month
```

将提取系统日期，然后将其转化为字符串，再显示月份的值。

技巧 179 使用 CURRENT_TIME 函数读取当前系统时间

每种 DBMS 都提供一种提取当前系统时间的方法。SQL 标准指明，CURRENT_TIME 函数当使用以下句法调用时将作为 TIME 数据类型返回系统时间：

```
CURRENT_TIME (<decimal second precision>)
```

由此

```
PRINT CURRENT_TIME(2)
```

将显示当前系统时间如下：

```
20:08:01.22
```

某些 DBMS 产品，如 DB2，将 CURRENT_TIME 实现为 TIME 数据类型的寄存器（也就是内存位置）。其他产品，如 MS-SQL Server 将作为部分函数（如 GETDATE()）调用返回系统时间。因而必须检查系统手册，找出具体的 DBMS 提供用来从操作系统中提取当前时间的准确变量或函数调用。例如，在 MS-SQL Server 上，以下语句：

```
PRINT GETDATE()
```

将与当前日期一起显示当前系统时间，其形式如下：

```
May 9 2000 8:15PM
```

然后既可使用 SUBSTRING 函数来提取由 GETDATE() 函数返回的 DATETIME 值的时间部分，也可使用在 CONVERT 函数只返回时间值。例如，以下语句：

```
PRINT CONVERT(VARCHAR, GETDATE(), 14)
```

将以以下形式只显示当前系统时间：

```
20:21:56.450
```

技巧 180 使用 CURRENT_DATE 函数读取当前系统日期

每种 DBMS 都提供一种提取当前系统日期的方法，如 SQL 的 CURRENT_DATE 函数，当执行以下语句时：

```
PRINT CURRENT_DATE
```

将显示以下格式的系统日期：

```
2000-09-04
```

某些 DBMS 产品，如 DB2，将 CURRENT_DATE 实现为 DATE 数据类型的寄存器（也就是内存位置）。其他产品，如 MS-SQL Server 将作为部分函数（如 GETDATE()）调用返回系统

日期。因而必须检查系统手册，找出具体的 DBMS 提供用来从操作系统中提取当前日期的准确变量或函数调用。例如，在 MS-SQL Server 上，以下语句：

```
PRINT GETDATE()
```

将与当前日期一起显示当前系统时间，其形式如下：

```
Sep 4 2000 9:19PM
```

然后既可使用 SUBSTRING 函数来提取由 GETDATE() 函数返回的 DATETIME 值的日期部分，也可使用 CONVERT 函数只返回日期值。例如，以下语句：

```
PRINT CONVERT (VARCHAR, GETDATE(), 106)
```

将只显示当前日期如下：

```
04 Sep 2000
```

技巧 181 使用 CURRENT_TIMESTAMP 函数读取当前系统日期和时间

大多数 DBMS 产品支持 CURRENT_TIMESTAMP 函数，此函数允许作为 TIMESTAMP 数据类型的单一值提取当前系统日期与时间。根据 SQL 标准，TIMESTAMP 数据类型定义为：

```
<date value><space><time value>
```

由此，以下语句：

```
PRINT CURRENT_TIMESTAMP
```

将以以下形式提取并显示当前系统日期和时间：

```
2000-09-05 05:40:01:547
```

某些 DBMS 产品，如 DB2，将 CURRENT_TIMESTAMP 实现为 TIMESTAMP 类型的寄存器（命名的内存位置）。在 MS-SQL Server 上，CURRENT_TIMESTAMP 函数与 GETDATE() 等价并返回 DATETIME 数据类型而不是 TIMESTAMP 类型的值。例如，在 MS-SQL Server 上执行以下语句：

```
PRINT 'The current TIMESTAMP is: ' +  
      CONVERT (VARCHAR, CURRENT_TIMESTAMP) +  
      ' and the GETDATE() value is: ' +  
      CONVERT (VARCHAR, GETDATE()) + '.'
```

将显示以下结果：

```
The current TIMESTAMP is: Sep 5 2000 6:54AM and the  
GETDATE() value is: Sep 5 2000 6:54AM
```

注意：DATETIME 数据类型是在 MS-SQL Server 上与 SQL 的 TIMESTAMP 等价的数据类型。这两种数据类型都用于定义保存以空格分开的日期和时间值的列和变量。

技巧 182 理解 MS-SQL Server 的日期和时间函数

与某些 DBMS 产品不同，MS-SQL Server 并不支持分开的 TIME、DATE 和 TIMESTAMP 数据类型，而是支持单一的 DATETIME 数据类型，用于定义保存复合的日期和时间值的列和变量。事实上，MS-SQL Server 甚至不支持分开的提取当前系统时间或当前系统日期的函数。Transact-SQL 的 GETDATE() 函数总是作为 DATETIME 类型的值返回系统日期和时间两者。

但是，可使用 CONVERT 函数将当前系统日期和时间值改变为只包括日期或只包括时间的字符串。由此，与 GETDATE() 结合使用 CONVERT 函数可用于生成与 CURRENT_TIME 和 CURRENT_DATE 函数同样的结果。但请记住，当用于从日期-时间值中提取日期或时间时，

CONVERT 函数必须返回 CHARACTER 类型的值而不是 DATE 或 TIME 类型的值（因为在 MS-SQL Server 上未定义 DATE 或 TIME 类型）。

表 10.2 列出的是当显示或存储当前时间时，MS-SQL Server 所使用的格式。例如，使用 CURRENT_TIME 函数将 TIME 类型的值以形式 15:25:23 放入 ORDERS 表的 TIME_ORDER_PLACED 列的语句：

```
INSERT INTO orders (time_order_placed)
VALUES (CURRENT_TIME)
```

其等价的 MS-SQL Server 语句为：

```
INSERT INTO orders (time_order_placed)
VALUES (CONVERT (VARCHAR, GETDATE (), 8))
```

表 10.2 使 GETDATE()函数与 CURRENT_TIME 函数近似的 CONVERT 函数的<style>值

<style>	格式	示例
8	hh:mm:ss	10:13:23
14	hh:mm:ss:sss	10:13:23:060

其中使用了 GETDATE()函数提取当前系统日期和时间（作为 DATETIME 类型），然后使用 CONVERT 函数将日期-时间值的时间部分提取到要放到 TIME_ORDER_PLACED 列的字符串中。

虽然 MS-SQL Server 不支持 CURRENT_DATE 函数，但可指定 CONVERT 函数中的<style>参数为表 10.3 中<style>值之一，从而使该函数将日期-时间表达式的日期部分转化为表示当前系统日期的字符串。例如，使用 CURRENT_DATE 值以形式 09/05/2000 放入 ORDERS 表的 DATE_ORDER_PLACED 列的以下语句：

```
INSERT INTO orders (date_order_placed)
VALUES (CURRENT_DATE)
```

其等价的 MS-SQL Server 语句为：

```
INSERT INTO orders (date_order_placed)
VALUES (CONVERT (VARCHAR, GETDATE (), 101))
```

表 10.3 使 GETDATE()函数与 CURRENT_DATE 函数近似的 CONVERT 函数的<style>值

<style>	格式	示例
1	mm/dd/yy	06/30/99
2	yy.mm.dd	99.06.30
3	dd/mm/yy	30/06/99
4	dd.mm.yy	30.06.99
5	dd-mm-yy	30-06-99
6	dd mmm yy	30 Jun 99
7	mmm dd, yy	Jun 30, 99
10	mm-dd-yy	06-30-99
11	yy/dd/mm	99/06/30
12	yymmdd	990630

其中使用了 GETDATE() 函数提取当前系统日期和时间（作为 DATETIME 类型），然后使用 CONVERT 函数将日期-时间值的日期部分提取到要放到 DATE_ORDER_PLACED 列的字符串中。

注意：在表 10.3 列出的 <style> 数上加 100 就告诉 MS-SQL Server 返回四位数的年而不是两位数的年。例如，CONVERT(VARCHAR, GETDATE(), 6) 返回日期为 30 Jun 99，而 CONVERT(VARCHAR, GETDATE(), 106) 返回同一日期为 30 Jun 1999。

最后，正如表 10.4 所示，可在 CONVERT 函数调用中使用 <style> 参数来控制 MS-SQL Server 如何提供当前系统的时间印记（日期和时间一起作为单个值）。例如，为了将时间印记以格式 2000-09-05 15:23:32.015 放入 ORDERS 表的 ORDER_TIMESTAMP 列，可执行以下语句：

```
INSERT INTO orders (order_timestamp)
VALUES (CONVERT(VARCHAR, CURRENT_TIMESTAMP, 21))
```

或者，使用 GETDATE() 函数代替 CURRENT_TIMESTAMP 函数，如下所示：

```
INSERT INTO orders (order_timestamp)
VALUES (CONVERT(VARCHAR, GETDATE(), 21))
```

以上语句可将同样的系统日期-时间值放入 ORDERS 表的一行的 ORDER_TIMESTAMP 列中。

表 10.4 用于转换 DATETIME 数据的 CONVERT 函数的 <style> 值

<style>	格式	示例
(blank), 0	mmm dd yyyy hh:mmAM/PM	Jun 30 1999 10:13AM
9	mmm dd yyyy hh:mm:ss:ssAM/PM	Jun 30 1999 10:13:23:060AM
13	dd mmm yyyy hh:mm:ss:sss	30 Jun 1999 10:13:23:060
20	yyyy-dd-mm hh:mm:ss	1999-06-30 10:13:23
21	yyyy-dd-mm hh:mm:ss.sss	1999-06-30 10:13:23.060

注意：如果从一个有 DATE 和 TIME 数据类型的系统转移代码，当想要在 MS-SQL Server 上只存储日期或只存储时间时，则必须使用 CHARACTER 类型的列。如果向数据类型为 DATETIME 的列中只放入日期或只放入时间，MS-SQL Server 将仍然在该列中既存储日期又存储时间值（如果忽略时间，MS-SQL Server 将 DATETIME 类型列的时间部分设置为 00:00:00.000；如果忽略日期，MS-SQL Server 将 DATETIME 类型列的日期部分设置为 1900-01-01）。

技巧 183 使用 CAST 函数将值从一种数据类型转化为另一种

MS-SQL Server 与 SQL-92 规范不同，当遇到在表达式中混合不同数据类型的值时，MS-SQL Server 是非常能够“容忍”的。根据 SQL 标准，DBMS 只应该自动转化那些类似数据类型的值。比如要将 SMALLINT 值与 INTEGER 值相比较，DBMS 应该执行该运算，但如果试图将字符串与 NUMERIC 类型的值相比较，DBMS 应该不允许比较并生成错误。

如果具体的 DBMS 遵循 SQL 标准的有关在表达式中结合不同类型数据的限制性规则，则可使用 CAST 函数将数据从一种类型转化为另一种，使得 DBMS 可以执行混合数据类型的表达式而没有错误。例如，假设在 ORDERS 表中将 CUSTOMER_ID 列作为 CHARACTER 字符存储，但在 CUSTOMERS 表中将 CUSTOMER_NUM 列作为 INTEGER 存储。只要 ORDERS

表的 CUSTOMER_ID 列中所有的 CHARACTER 数据类型值是 NULL 或所有都是数值位，就可在 SELECT 语句中使用 CAST 函数将 CHARACTER 类型的 CUSTOMER_ID 转化为 INTEGER 类型，如下所示：

```
SELECT customer_name, product_code, qty, amount
FROM customers, orders
WHERE CUSTOMER_NUM = CAST (customer_ID INTEGER)
ORDER BY customer_name
```

这样一来，DBMS 就能够将 INTEGER 值与查询的 WHERE 子句中的另一个 INTEGER 值相比较，而不是试图将 INTEGER 类型的 CUSTOMER_NUM 列与 CHARACTER 类型的 CUSTOMER_ID 相比较而生成错误。

由于大多数 DBMS 产品都不是严格地遵循 SQL 标准的严谨数据类型检查规则，当需要从 DBMS 向不支持某种值的数据类型的应用程序中传递值时常常要使用 CAST 函数。例如，大多数编程语言不支持 DATETIME 数据类型。因而，必须使用 CAST 函数将 DATETIME 数据类型的值转化为 CHARACTER 字符串，以便主调程序可以加以处理。例如，下面的 SELECT 语句：

```
SELECT customer_ID, CAST (order_date AS CHAR(11)),
CAST(date_shipped as CHAR(11)), order_total FROM Orders
```

将把 ORDER_DATE 和 DATE_SHIPPED 列的 DATETIME 类型的值转化为 10 个字母的字符串（如果 SELECT 语句将其结果放入临时表，此内容将在技巧 287 “使用 SQLFetch 函数从 SQL 数据库中提取数据行” 中学习有关内容，然后就可将 ORDER_DATE 和 DATE_SHIPPED 作为 CHARACTER 类型的字符串而不是主调程序不支持的 DATETIME 值传递到主调程序变量中）。

正如本例中所表明的，CAST 函数调用的句法如下：

```
CAST (<value expression> AS <data type>)
```

可将其用于 SQL 语句中任何 <data type> 可出现的地方（类似于使用其他诸如 CONVERT or GETDATE() 一类的函数的方式）。请记住，只能使用 CAST 函数执行合法的数据类型转换。

例如，DBMS 不能执行以下语句：

```
PRINT CAST('Hello' AS INTEGER)
```

因为 CHARACTER 类型的实际值包括非数字字符。但是以下语句将可成功执行：

```
PRINT CAST('-12.35' AS INTEGER)
```

因为字符实际值包括的都是数字、零、一个小数点、一个加号 (+)、零、一个减号 (-)。简短地说，如果 DBMS 允许向由 <data type> 给定的列中输入特定的实际值，那么就可使用以下形式的 CAST 函数把 <value expression> 中的值转换为等价的 <data type> 类型的值：

```
CAST (<value expression> AS <data type>)
```

注意：当把数值型的值转换为 CHARACTER 字符串时，请确保字符串足够大到存储所有的数字、小数点（如果有的话）和符号（如果数值为负的话）。如果系统转换后要放入的字符串太短，不能保存其值的所有部分，某些 DBMS 产品将截短数字以便能够放入字符串，而另一些产品，如 MS-SQL Server，将中止语句的执行并出现如下错误：

```
Server: Msg 8115, Level 16, State 5, Line 1
Arithmetic overflow error converting numeric to data type
varchar.
```

技巧 184 使用 CASE 表达式根据列的值选择实际值

CASE 表达式为 SQL 提供了有限的决策能力。正如以前所提到的, SQL 是数据子语言而不是完全编程的或面向对象的编程语言。SQL 只能允许告诉 DBMS 要找什么或做什么, 而不是如何找或如何做。大多数 DBMS 产品对标准的 SQL 提供扩展, 使其允许对 SQL 批处理中的语句的执行顺序编程。同时 CASE 表达式是 SQL 本身的一部分并给出数据子语言的 IF-THEN-ELSE 控制结构, 可用来根据对其操作的表列中的数据来决定哪一条 SQL 语句将要执行。

当 DBMS 遇到一条以下形式的 CASE 语句时:

```
CASE WHEN <search condition> THEN <expression>
[...WHEN <last search condition>
THEN <last expression>]
[ELSE <expression>]
END
```

系统检查第一个<search condition>, 如果其求值为 TRUE, 那么 CASE 语句取第一个<expression>的值; 如果第一个<search condition>求值为 FALSE, 那么 DBMS 继续对第二个<search condition>求值; 如果第二个<search condition>求值为 TRUE, 那么 CASE 语句取第二个<expression>的值; 如果为 FALSE, DBMS 继续检查第 3 个<search condition>, 如此依此类推。如果没有 CASE 表达式的搜索条件求值为 TRUE, 那么 CASE 就取其 ELSE 子句中的表达式的值, 或者如果没有 ELSE 子句, 则 CASE 表达式求值为 NULL。

例如, 以下 SELECT 语句中的 CASE 表达式:

```
SELECT first_name, last_name,
CASE WHEN pts_accumulated >= 100000 THEN 'High Roller'
      WHEN pts_accumulated >= 50000 THEN 'Premium Player'
      WHEN pts_accumulated >= 25000 THEN 'Preferred Guest'
      WHEN pts_accumulated >= 12500 THEN 'Regular'
      ELSE 'Patron'
END AS 'Rating',
pts_accumulated
FROM casino_guests
```

可用于在以下结果表中显示展会客人姓名和游戏者的等级:

first_name	last_name	Rating	pts_accumulated
Sally	Fields	Patron	1300
Lenny	Hall	Premium Player	50000
Wally	Wells	Regular	13000
Brent	McCoy	Preferred Guest	25000
Erwin	Shiff	Preferred Guest	45000
Bruce	Wayne	Premium Player	75000
James	Bond	High Roller	107500

正如本例所表明的那样, DBMS 对第一个搜索条件 pts_accumulated >=100000 求值, 如果为 TRUE, 则将结果表中 RATING 列的值设置为 High Roller。如果搜索条件求值为 FALSE, DBMS 移向下一搜索条件, 直到找到一个求值为 TRUE 为止。如果没有搜索条件求值为 TRUE, 正如在本例中的 Sally 的情况, DBMS 将结果表中 RATING 列的值设置为 ELSE 子句中的表达

式的值。

技巧 185 在搜索的 CASE 表达式中使用子查询

在技巧 184 “使用 CASE 表达式根据列的值选择实际值”中，读者已经学习了如何使用 CASE 表达式的简单形式，在此形式中，WHEN 子句是由列引用组成的，而且所有的 THEN 子句的值（结果表达式）都是实际值。但是，除了列名和实际值之外，还可在 CASE 表达式的子句中使用任何合法的 SQL 表达式。例如，假设读者是一位废金属经销商，而且如果手中的存货量低于想要的量时，将要在基价之上再支付 25% 的保险费。通过在下面查询的 CASE 表达式中包括子查询：

```
SELECT metal, desired_lbs,
       (SELECT SUM(lbs) FROM scrap_inv
        WHERE scrap_inv.metal = scrap_master.metal) AS
       'lbs_on_hand',
       base_price,
       CASE WHEN (SELECT SUM(lbs) FROM scrap_inv
                  WHERE scrap_inv.metal = scrap_master.metal) <=
                 desired_lbs THEN base_price * 1.25
              ELSE base_price
       END AS 'current_price'
FROM scrap_master
```

可在 WHEN 子句中使用 SUM 总计函数找出当前在手的金属量。然后子查询的结果决定结果表中 CURRENT_PRICE 的值。如果费率正处于或低于所希望的水平，THEN 子句将当前价格设置为 SCRAP_MASTER 列中的基价的 125%，从而产生如下的结果表：

metal	desired_lbs	lbs_on_hand	base_price	current_price
Tin	1000	1625	.7500	.750000
Iron	1500	625	.2500	.312500
Aluminum	2000	850	.4500	.562500

除了在 CASE 表达式的 WHEN 子句中使用子查询之外，还可在 THEN 子句中使用子查询。例如，假设当存货量小于想要的量时，将当前价格设置为在以前支付的平均价上加付 25%。有如下 CASE 表达式的 SELECT 语句：

```
SELECT metal, desired_lbs,
       (SELECT SUM(lbs) FROM scrap_inv
        WHERE scrap_inv.metal = scrap_master.metal) AS
       'lbs_on_hand',
       base_price,
       CASE WHEN (SELECT SUM(lbs) FROM scrap_inv
                  WHERE scrap_inv.metal = scrap_master.metal) <=
                 desired_lbs
       THEN CASE WHEN (SELECT SUM(lbs) FROM scrap_inv
                       WHERE scrap_inv.metal =
                         scrap_master.metal) > 0
                THEN (SELECT AVG(purchase_price) * 1.25
```

```

        FROM scrap_inv
        WHERE scrap_inv.metal =
              scrap_master.metal)
        ELSE base_price * 1.25
      END
    ELSE base_price
  END AS 'current_price'
FROM scrap_master

```

将生成当手中的货量低时根据金属的平均价而不是根据其基价计算的 CURRENT_PRICE 的结果表。

技巧 186 使用 NULLIF 函数将列值设置为 NULL

NULLIF 函数是 CASE 表达式的特殊形式, 如果作为参数传递到函数中的两个表达式有相等的值, 允许人们将列值设置为 NULL。当 DBMS 遇到以下形式的 NULLIF 函数时:

```
NULLIF (<first expression>, <second expression>)
```

提取<first expression> (通常为列值) 并将其值与<second expression>表达式 (通常为数值型或字符串实际值) 的值相比较。如果两个表达式求值一样 (相等), 那么 NULLIF 函数返回 NULL 值。否则, 函数返回<first expression>表达式的值。

如果由列值所代表的物理实体或概念的属性从已知变为未知或是缺少状态, 就可以使用 NULLIF 函数。例如, 假设有一个 EMPLOYEES 表, 其中 MANAGER 列包括雇员经理的 ID (EMP_ID)。如果经理 Susan (EMP_ID 号为 101) 离开了公司, 她管理的雇员就不再有经理了。现在可使用以下带 CASE 表达式的 UPDATE 语句将原 Susan 管理的所有雇员的 MANAGER 列设为 NULL:

```

UPDATE employees SET manager =
  CASE WHEN manager = 101 THEN NULL
        ELSE manager
  END

```

或者, 执行下面带 NULLIF 函数的 UPDATE 语句达到同样的目的:

```
UPDATE employees SET manager = NULLIF(manager, 101)
```

在本例中, 如果行的 MANAGER 列的值为 101, 那么 NULLIF 函数返回 UPDATE 语句放入 MANAGER 列的 NULL 值。否则, NULLIF 函数返回 MANAGER 列中的值——其意义为 UPDATE 语句将保持那些 MANAGER 列值不为 101 的所有行不变。

当导入以前由一种标准的编程语言维护的数据时, NULLIF 函数特别有用。例如, NULL 值在 COBOL、Pascal 和 FORTRAN 这类语言中是不允许的。结果, 编程人员可能在诸如 SALARY 这样的列中使用了特殊的代码-1 (比如说) 来表明对于那些以小时支付工资的人的 NULL 工资。为了将 SALARY 列中的-1 值转换为 NULL, 可执行以下带 CASE 表达式的 UPDATE 语句:

```

UPDATE employees SET salary = CASE salary WHEN -1 then NULL
                                   ELSE salary
  END

```

或者, 可执行更为紧凑的 NULLIF 函数, 如下所示:

```
UPDATE employees SET salary = NULLIF(salary, -1)
```

技巧 187 使用 CAST 函数比较不同数据类型列中的值

严格遵循 SQL-92 标准的 DBMS 产品不允许在单个表达式中混合使用不同数据类型的值。因而，遵循 SQL-92 指导的 DBMS 产品会中止以下 SELECT 语句：

```
SELECT employees WHERE hire_date = 'Aug 20 2000'
```

因为 WHERE 子句中的 BOOLEAN 表达式试图将 DATETIME 类型的列值与字符串实际值相比较。

幸运的是，许多 DBMS 产品，如 MS-SQL Server，将自动地将本例 WHERE 子句中的两个值转换为共同的数据类型，然后再执行语句，而不出现错误。但是，如果 DBMS 仅当操作数涉及类似的数据类型（如 INTEGER 和 SMALLINT 或 FLOAT 和 NUMERIC 或 CHAR 和 VARCHAR）时才执行类型转换，则可使用 CAST 函数执行手工转换。

例如，如果具体的 DBMS 不允许将字符串实际值（字符串）与 DATETIME 类型的列值比较，可以如下形式使用 CAST 函数将 DATETIME 类型值转换为字符串：

```
CAST (<value expression> AS <data type>)
```

因而，以下查询 DBMS 是可接受的：

```
SELECT * FROM employees  
WHERE CAST(hire_date AS CHARACTER(11)) = 'Aug 20 2000'
```

因为在 WHERE 子句中的两个值是字符串。类似地，如果有一 EMPLOYEE 表，其中 EMP_ID 列是 INTEGER 类型而 CUSTOMER 表中的 SALESPERSON_ID 列是 VARCHAR(6) 类型，在以下查询中 CAST 函数将把 SALESPERSON_ID 转换为 INTEGER 类型的值：

```
SELECT customer_name, employee_name AS 'sold_by'  
FROM employees, customers  
WHERE emp_ID = CAST(salesperson_ID AS INTEGER)
```

这样一来，DBMS 将可将其与 EMP_ID 中的值相比较而不会出现错误。简短地说，在 SQL 语句中任何表达式或值可出现的地方都可使用 CAST 表达式。当然，对 CAST 表达式可做的是某些限制。例如，不能使用 CAST 将字符串 Konrad 转换为 MONEY 类型的值，毕竟 Konrad 中没有任何数字。类似地，不能使用 CAST 将 09/19/2000 转换为 INTEGER 类型的值。

基本上说，可使用 CAST 将任何数据类型转换为 CHARACTER 或 VARCHAR 类型。但是，仅当非数值类型的值中包括的全部是数字（0-9）、（可选的）单个小数点和（可选的）字符串开头的单个横杠（-）或加号（+），用于说明数字是正还是负，才能将其转换为数值类型。

技巧 188 使用 CAST 函数从 SQL 向宿主语言中传递值

宿主变量允许将值从用宿主编程语言（如 C、FORTRAN、Pascal 或 Visual Basic）编写的应用程序中向 DBMS 值以及将值从 DBMS 传回到宿主程序。遗憾的是，没有一种编程语言支持所有的 SQL 的数据类型。CAST 函数允许将宿主语言不支持的 SQL 数据类型的值转换为等价的 DATETIME 数据类型（比如说），这样就可将日期时间的组合值存储在 SQL 的表列中。但是，没有一种宿主编程语言支持 DATETIME 类型。因而必须将想要传递到外部程序中的 DATETIME 类型的值转换为程序支持的数据类型。这正是 CAST 函数大有用武之地的地方。

假设想将雇员的雇用日期传递到 C 程序中。由于 C 不支持 DATETIME 类型，此时可在下面的语句中使用 CAST 函数将日期与时间的组合信息转换为字符串值：

```
SET :date_hired = (SELECT CAST(hire_date AS CHAR(19))
FROM employees
WHERE employee_id = 101)
```

在本例中，C 程序的字符串分析子程序将从日期与时间字符串中提取雇用日期，以便使用标准的 C 的 I/O 函数将结果显示在屏幕上。

在技巧 280–技巧 297 中，读者将学习在宿主（编程的）语言应用程序中嵌入 SQL 命令的方法。现在，要了解的重要事情是，CAST 函数给出了一种将 SQL 中独一无二的数据类型值通过宿主变量从 DBMS 传递到宿主程序中的方法。

技巧 189 理解在 Select 语句中如何使用修饰子句

SELECT 语句可用来读取并显示 SQL 表列中的数据值。修饰子句 FROM、WHERE、HAVING、GROUP BY 和 ORDER BY 允许指定要查询的表、要显示哪些数据以及如何对结果行排序。表 10.5 总结了可用于 SELECT 语句中的每个修饰子句的目的。

表 10.5 Select 语句的修饰子句

子句	目的
FROM	指定 SELECT 语句要显示其中列值数据的表
WHERE	指定用来筛选查询不显示的行的搜索条件
GROUP BY	根据组合列中的值将行组合成一组
HAVING	筛选不满足一个或多个搜索条件的组。（HAVING 只能与 GROUP BY 子句一起出现）
ORDER BY	根据指定的列（只能引用 SELECT 子句中列出的列）对结果表中的行排序（在显示之前）

至少，每条 SELECT 语句必须有一条 FROM 子句（毕竟，必须告诉 DBMS 要查询哪个表或哪几个表）。除了 FROM 子句之外，查询可按以下顺序包括任何或所有修饰子句：

```
SELECT <column list>
FROM <table list>
[WHERE <search condition(s)>]
[GROUP BY <grouping column list>]
[HAVING <search condition(s)>]
[ORDER BY <column list>]
```

例外的是 FROM 子句（用于指定 SELECT 语句要操作的表），DBMS 处理其在查询中所找到的结果以顺序方式进行。换句话说，一个子句的结果表作为下一子句的输入表。

例如，以下的查询创建由 CUSTOMERS 表中的行与 INVOICES 表中的行组成的笛卡尔积的结果表：

```
SELECT customers.cust_ID, cust_name, inv_total
FROM customers, invoices
添加 WHERE 子句：
SELECT customers.cust_ID, cust_name, inv_total
FROM customers, invoices
WHERE customers.cust_ID = invoices.cust_ID
```

告诉 DBMS 筛选出 CUSTOMERS 表中那些 CUST_ID 不等于 INVOICES 表中的 CUST_ID 列的行。

添加 GROUP BY 子句:

```
SELECT customers.cust_ID, cust_name, SUM(inv_total)
FROM customers, invoices
WHERE customers.cust_ID = invoices.cust_ID
GROUP BY customers.cust_ID, cust_name
```

告诉 DBMS 使用由 WHERE 子句生成的结果表的行与具有相同复合 (CUST_ID、CUST_NAME) 值的所有行组合成单一行。

添加 HAVING 子句:

```
SELECT customers.cust_ID, cust_name, SUM(inv_total)
FROM customers, invoices
WHERE customers.cust_ID = invoices.cust_ID
GROUP BY customers.cust_ID, cust_name
HAVING customers.cust_ID > 500
```

告诉 DBMS 将从 GROUP BY 子句得来的结果表进行筛选, 选出那些 CUST_ID 列小于 500 的行。

最后添加 ORDER BY 子句:

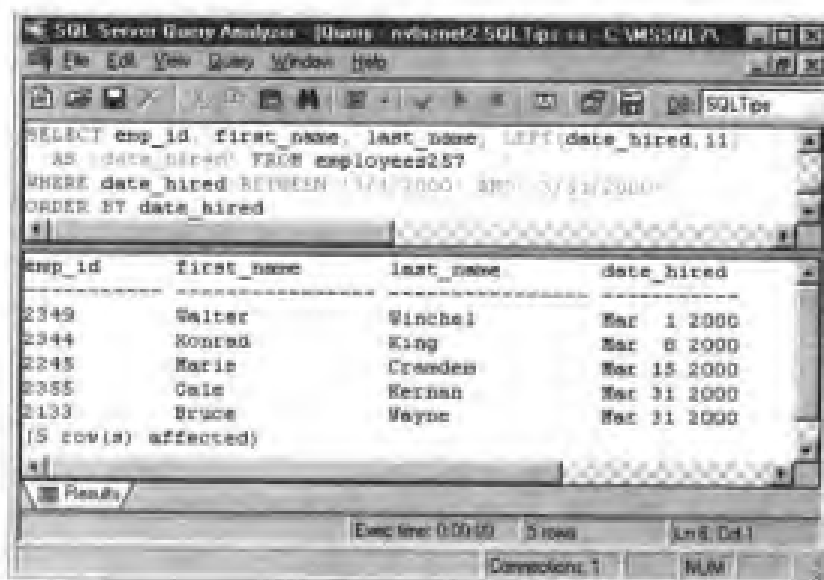
```
SELECT customers.cust_ID, cust_name, SUM(inv_total)
FROM customers, invoices
WHERE customers.cust_ID = invoices.cust_ID
GROUP BY customers.cust_ID, cust_name
HAVING customers.cust_ID > 500
ORDER BY cust_name
```

告诉 DBMS 将由 HAVING 子句生成的结果表中的行根据每行的 CUST_NAME 列的值以升序排列。

第 11 章 使用比较判式和组合查询

技巧 190 在 WHERE 子句中使用 BETWEEN 关键词选择行

当想要处理列值处在特定的值范围内的几行时，可在语句的 WHERE 子句中使用 BETWEEN 关键词。例如，为了获得在 2000 年 3 月中雇用的雇员清单，可使用类似于图 11.1 中 MS-SQL Server Query Analyzer 的输入窗格中靠近顶部的 SELECT 语句来产生类似于显示于该图底部的结果表。



The screenshot shows the MS-SQL Server Query Analyzer interface. The top pane contains the following SQL query:

```
SELECT emp_id, first_name, last_name, LEFT(date_hired, 11)
AS 'date_hired' FROM employees257
WHERE date_hired BETWEEN '3/1/2000' AND '3/31/2000'
ORDER BY date_hired
```

The bottom pane displays the results of the query in a table format:

emp_id	first_name	last_name	date_hired
2349	Walter	Winchel	Mar 1 2000
2344	Konrad	King	Mar 8 2000
2345	Marie	Crawden	Mar 15 2000
2385	Gale	Kerrnan	Mar 31 2000
2133	Bruce	Wayne	Mar 31 2000

Below the table, it indicates "(5 row(s) affected)". At the bottom of the window, the status bar shows "Exec time: 0:00:00", "5 rows", "Ln 6, Col 1", and "Connections: 1".

图 11.1 MS-SQL Server Query Analyzer 使用 BETWEEN 关键词的查询和结果表

虽然本例中的 SELECT 语句展示的是使用范围的上限和下限是实际值的查询，但 BETWEEN 判式实际上是由任何 3 个带有兼容数据类型的 SQL 表达式组成，其句法如下：

<test expression>

BETWEEN <low expression> AND <high expression>

例如，可使用以下 SELECT 语句列出那些总销售额减去 \$25,000 之后处于平均销售额与平均销售额的 120% 之间的雇员：

```
SELECT * FROM employees
```

```
WHERE (total_sales - 25000)
```

```
BETWEEN (SELECT AVG(total_sales) FROM employees)
```

```
AND (SELECT AVG(total_sales) * 1.2 FROM employees)
```

只要 <test expression> 的值大于或等于 <low expression> 的值且小于或等于 <high expression> 的值，则 BETWEEN 判式求值为 TRUE。因而下面的查询：

```
SELECT first_name, last_name FROM employees
```

```
WHERE last_name BETWEEN 'J' and 'Qz'
```

与以下查询是等价的：

```
SELECT first_name, last_name FROM employees
WHERE last_name >= 'J' and last_name <= 'Qz'
```

范围的低端值（<low expression>）必须小于或等于范围的高端值（<high expression>）。

另外要注意：

```
SELECT * FROM employees WHERE date_hired
BETWEEN '01/01/2000' AND '01/31/2000'
```

看起来似乎与以下查询是等价的：

```
SELECT * FROM employees WHERE date_hired
BETWEEN '01/31/2000' AND '01/01/2000'
```

实际上并非如此。第一个查询列出的是在 2000 年 1 月雇用的雇员，而第二个查询不会列出任何雇员，因为不能出现 DATE_HIRED 大于或等于 01/31/2000（<low expression>）同时又小于或等于 01/01/2000（<high expression>）的情况。

技巧 191 在 WHERE 子句中使用 IN 或 NOT IN 判式选择行

IN 和 NOT IN 判式允许根据一行中是否有一个列值处于（包括在）一系列值之中而选择行。例如，假设公司在 Nevada、California、Utah 和 Texas 州各有一个办公室。由于必须向居住在那些州的顾客征收销售税，可在以下 UPDATE 语句中使用 IN 判式来计算销售税（假设每个州的税率均为 7%）：

```
UPDATE invoices SET sales_tax = invoice_total * 0.07
WHERE ship_to_state IN ('NV', 'CA', 'UT', 'TX')
```

作为对照，以下 UPDATE 语句中的 NOT IN 判式将把那些居住在公司未设办公室的州中的顾客的税率设置为 0.00：

```
UPDATE invoices SET sales_tax = 0.00
WHERE ship_to_state NOT IN ('NV', 'CA', 'UT', 'TX')
```

IN 和 NOT IN 判式的句法如下：

```
<test expression>
[NOT] IN (<first value> [... , <last value>])
```

当<test expression>处于在括号（）中列出的一系列值之中时，IN 判式求值为 TRUE，当<test expression>不处于该系列值之中时，NOT IN 判式则求值为 TRUE。

技巧 192 在 LIKE 判式中使用通配符

当只知道部分字符串时，可使用 LIKE 判式来查询数据库找出整个字符串。由此，当把关键词 LIKE 用在 WHERE 子句中时，允许比较两个字符串的部分匹配，当对字符串内容有某种印象，并不知道准确的形式时，这种对部分匹配的比较特别有价值。

LIKE 判式有两个通配符，当仅知道其通用形式而不知道其所有字符时，可用来编写搜索字符串。百分符（%）可代表长度为 0 个或多个字符的任何字符串，而下划线（_）可代表任何单个字符。为了使用一个或多个通配符在 LIKE 查询中搜索字符串，只要在字符串实际值中与已知部分一起包括通配符。

例如，为了搜索那些姓（last_name）以字符“Ki”开始的所有教师，可执行以下 SELECT 语句：

```
SELECT first_name, last_name FROM faculty
```

```
WHERE last_name LIKE 'K%'
```

将产生下面的结果表:

```
first_name last_name
```

```
-----
```

```
Konrad      King
```

```
Wally       Kingsly
```

```
Sam         Kingston
```

从 FACULTY 表中选出的每行都有以两个字母 “Ki” 开始的 LAST_NAME 列, 后面跟随有任意数目的其他字符。

类似地, 如果想要匹配任何单个字符而不是多个字符, 可以使用下划线 (_) 通配符, 而不是百分符 (%)。例如, 如果知道字母 S 是 Sciences 课表中的任何课程中的第二个字母, 而且想要得到 100 类课程的清单, 即可执行以下 SELECT 语句:

```
SELECT course_ID, description FROM curriculum
```

```
WHERE course_ID LIKE '_S10_'
```

用来产生以下结果表:

```
course_ID description
```

```
-----
```

```
CS101      Introduction to Computer Science
```

```
BS109      Biological Sciences - Anatomy & Physiology
```

```
MS107      Beginning Quadratic Equations
```

DBMS 从 CURRICULUM 表中只选择那些 COURSE_ID 值中 S 为第二个字符, 后面跟有 10, 而且以一个且只有一个另外的字符结束的行。与百分符 (%) 通配符 (可匹配 0 个或多个字符) 不同, 下划线 (_) 通配符可匹配一个且只能匹配一个字符。因此, 在本例中, course_ID 为 S101 和 CS101H 不会包括在查询的结果表中。S101 中的 S 是第一个而不是第二个字符, 而在 CS101H 中, 10 后跟着两个而不是一个字符。

技巧 193 在 LIKE 判式中使用转义字符

为了检查在字符数据类型的列中是否存在百分符 (%), 需要有一种方法告诉 DBMS 将 LIKE 判式中的百分符看作实际值而不是通配符。关键词 ESCAPE 允许确定一个转义字符, 告诉 DBMS 将搜索字符串中紧跟在转义字符之后的字符看作实际值。例如, 以下查询:

```
SELECT cust_ID, cust_name, discount FROM customers
```

```
WHERE discount LIKE "%S%" ESCAPE 'S'
```

使用转义字符 S 告诉 DBMS 将搜索字符串 %S% 中的第二个百分符 (%) 看作为实际值 (而不是通配符)。结果, 查询将返回那些在 DISCOUNT 列中的最后一个字符为百分符的行中的 CUST_ID、CUST_NAME 和 DISCOUNT 列值。

类似地, 如果想要在字符数据类型列中搜索下划线字符, 可在以下的查询中使用关键词 ESCAPE 告诉 DBMS 将紧跟在美元符 (\$) 后的字符看到为实际的字符值而不是通配符:

```
SELECT product_code, description FROM inventory
```

```
WHERE product_code LIKE "XY$_%" ESCAPE '$'
```

作为在本例中的查询结果, DBMS 将显示 INVENTORY 表中产品代码以 “XY_” 开始的所有 PRODUCT_CODE 和 DESCRIPTION 列值。

技巧 194 使用 LIKE 和 NOT LIKE 比较两个字符串

SQL 有两个允许搜索 CHARACTER、VARCHAR 或 TEXT 列的内容从而找出某种字符排列样式的判式。LIKE 判式返回目标列中包括搜索字符串中的字符排列样式的行集。作为对照，NOT LIKE 判式返回那些在目标列中没有找到给定字符排列样式的行集。

例如，以下查询将显示 CUSTOMERS 表中目标列（CUST_NAME）为 KING 的行：

```
SELECT * FROM customers WHERE cust_name LIKE 'KING'
```

而以下查询将显示 CUSTOMERS 表中在 CUST_NAME 列没有 SMITH 的那些行：

```
SELECT * FROM customers WHERE cust_name NOT LIKE 'SMITH'
```

如果不与读者在技巧 192“在 LIKE 判式中使用通配符”中学习过的通配符结合使用，LIKE 和 NOT LIKE 就没有什么大用。毕竟，上面的两个示例可用相等（=）和不等（<>）比较运算符来代替，如下所示：

```
SELECT * FROM customers WHERE cust_name = 'KING'
```

```
SELECT * FROM customers WHERE cust_name <> 'SMITH'
```

简短地说，当只“知道”目标列中的某些字符或是想要使用包括某种字符排列样式的所有时，LIKE 判式是有用的。例如，如果“知道”顾客名好像是 KING 什么的，但不能确定是 KINGSLEY、KING 还是 KINGSTON，就可使用以下 SELECT 语句加以查询：

```
SELECT * FROM customers WHERE cust_name LIKE 'KING%'
```

百分符（%）通配符告诉 DBMS 匹配任何 0 个或多个字符。因而在本例中的查询告诉 DBMS 显示 CUSTOMERS 表中的那些目标列（CUST_NAME）以字符 KING 开始后面跟有 0 个或多个另外的字符的行。

类似地，如果在搜索字符串前面加百分符，即可搜索字符串内的字符排列样式。例如，以下查询：

```
SELECT * FROM customers WHERE notes LIKE "%give%discount%"
```

将显示 CUSTOMERS 表中那些在目标列（NOTES）中包括单词 GIVE，后面至少出现一次 DISCOUNT 的行。因而，DBMS 将显示在一行中的 NOTES 包括字符串“Excellent customer. Make sure to give a 5% discount with next order.”的行。

作为对照，当与一个或多个通配符结合使用时，NOT LIKE 判式将显示那些不包括由搜索字符串给定的字符排列样式的行。这样一来，以下查询：

```
SELECT * FROM customers WHERE notes NOT LIKE "%discount%"
```

将显示那些 NOTES 列中不包括组成单词 discount 的字母排列样式的行。

技巧 195 理解 MS-SQL Server 对 LIKE 判式中的通配符的扩展

在技巧 192“在 LIKE 判式中使用通配符”中，读者已经学习了如何在 LIKE 判式中使用百分符和下划线通配符来比较字符串从而找出部分匹配。虽然下划线允许匹配任何单个字符，而百分符允许匹配 0 个或多个字符，但这两个通配符都不允许指定不知道的字符必须落入的字符范围。例如，以下查询：

```
SELECT * FROM employees WHERE badge LIKE '1___'
```

在 1 后有 3 个下划线，这就告诉 DBMS 显示那些雇员证号为四位数且以 1 开始的所有雇员。如果想要将结果限制为证件号为，其第一个字符为 1，（比如说）第二个字符为 a、A、b、

B、c 或 C，MS-SQL Server 允许使用方括号 ([]) 提供下划线 (_) 通配符加以匹配的字符集合。因此，以下查询：

```
SELECT * FROM employees WHERE badge LIKE '1[a-zA-C]__'
```

告诉 DBMS 显示 4 个字符的证件号，其中第一个字符为 1，第二个字符是大写或小写的 A、B、C，后面再跟任何其他字符（由搜索字符串中最后两个下划线所代表的）。

作为对照，如果想要从由通配符匹配的字符串中排除字符范围，可在左方括号 ([]) 和要代替通配符的字符范围的第一个值之间插入脱字字符 (^)。例如，如果想要得到证件号的第一个字符为 1~9，而其余 3 个字符不能是字母的雇员清单，MS-SQL Server 允许执行以下查询：

```
SELECT * FROM employees
WHERE badge LIKE '[1-9][^a-zA-Z][^a-zA-Z][^a-zA-Z]'
```

技巧 196 使用 NULL 判式找出所选列中有 NULL 值的所有行

列中的 NULL 值表明未知或是缺少的数据。由于列中的实际值是未知的，DBMS 不能对其值做任何假设。结果，下面的 SELECT 语句不会显示任何行：

```
SELECT * FROM employees WHERE manager = NULL
```

即使 EMPLOYEES 表中几行在 MANAGER 列有 NULL 值。

这种似乎不正确的行为，即 WHERE 子句中的测试：

```
NULL = NULL
```

当 MANAGER 列的值为 NULL 时不会返回 TRUE 的原因是，NULL 真正意义是“未知的”。因而，DBMS 不能对相等运算符左侧的未知值做出是否等于右侧的未知值的决定。所以，DBMS 对

```
NULL = NULL
```

只能返回 NULL 而不是 TRUE。

为了找出表列的值是否为 NULL，可使用 IS NULL 判式。与相等运算符不同（此运算符仅当运算符两侧的数据项相等时才能求值为 TRUE），IS NULL 运算符只是测试当时的状态，也就是说，“列中的值是 NULL 吗？”在没有对列中的实际值做任何假设的情况下，既可为 TRUE 也可为 FALSE。这样一来，以下查询将显示 EMPLOYEES 表中那些 MANAGER 列值为 NULL（未知或缺少）的行：

```
SELECT * FROM employees WHERE manager IS NULL
```

SQL 还提供一种特殊的判式，可用来找出那些在特定的列中没有 NULL 值的行。毕竟，如果测试：

```
NULL = NULL
```

求值为 NULL（而不是 TRUE），下面的测试：

```
NULL <> NULL
```

一定也会求值为 NULL，因为 DBMS 不能对不等号比较运算符 (<>) 两边的 NULL（未知）做任何假设。

为了查询那些在 MANAGER 列中的值不是 NULL 的行，可使用 IS NOT NULL 判式。例如，下面的 SELECT 语句将显示 EMPLOYEES 表中 MANAGER 列不是 NULL 的那些行：

```
SELECT * FROM employees WHERE manager IS NOT NULL
```

技巧 197 理解 UNIQUE 判式

WHERE 子句中的 UNIQUE 判式允许根据 UNIQUE 判式中的子查询是否生成所有行都不是重复的结果表（也就是说，结果表中的所有行都是惟一的）来执行 DELETE、INSERT、SELECT 或 UPDATE 语句。如果子查询的结果表的行是惟一的，则 DBMS 对正在测试的行执行 SQL 语句。作为对照，如果结果表至少有一对是重复的行，DBMS 跳过 SQL 语句的执行并转而检查表中的下一行。

例如，为了获得那些要么没有销售额，要么在 2000 年 9 月只有一项销售记录的销售人员的清单，可执行以下查询：

```
SELECT emp_ID, first_name, last_name FROM employees
WHERE UNIQUE (SELECT salesperson FROM invoices
               WHERE invoice_date >= '9/1/2000' AND
                  invoice_date <= '9/30/2000' AND
                  invoices.salesperson = employees.emp_ID)
```

UNIQUE 总是与一个子查询一起使用，其句法为：

```
[NOT] UNIQUE <subquery>
```

如果由 UNIQUE 判式中的子查询产生的结果表要么没有行，要么没有重复的行，则 UNIQUE 判式返回 TRUE。在本例中，子查询的结果表有一个单列 SALESPERSON。因而，如果销售人员的 ID 出现在结果表中不多于一行（即在所指定期间最多有一项销售记录的人），外层的 SELECT 语句将显示雇员的 ID 和姓名。另一方面，如果子查询的结果表有一或多个重复的行，UNIQUE 判式求值为 FALSE，则 DBMS 进而测试 EMPLOYEES 表的下一行。

注意：当在子查询的结果表中检查重复值时，UNIQUE 判式忽略任何 NULL 值。因此，如果 UNIQUE 判式中的子查询产生以下结果表：

Salesperson	cust_ID	sales_total
101	NULL	100.00
101	1000	NULL
101	NULL	NULL
101	1000	100.00

则判式求值为 TRUE，因为没有两行中所有的列都有相同的非 NULL 值。

几乎没有 DBMS 产品支持 UNIQUE 判式。因而，在将其用于代码中之前，一定要检查系统手册。如果具体的 DBMS 不支持此判式，总可以在 WHERE 子句中使用 COUNT 总计函数来完成相同的任务。例如，以下查询：

```
SELECT emp_ID, first_name, last_name FROM employees
WHERE (SELECT count(salesperson) FROM invoices
       WHERE invoice_date >= '9/1/2000' AND
          invoice_date <= '9/30/2000' AND
          invoices.salesperson = employees.emp_ID) <= 1
```

将产生与前例使用 UNIQUE 判式查询的相同结果，列出在 2000 年 9 月中只有 0 项或一项销售记录的销售人员。

技巧 198 使用 OVERLAPS 判式决定一个 DATETIME 是否与另一个重叠

OVERLAPS 判式使用以下句法:

```
{<temporal value 1>,  
{<temporal value 2>|<temporal argument>}} OVERLAPS  
{<temporal value 3>,  
{<temporal value 4>|<temporal argument>}}
```

其中:

```
<temporal value>::  
{DATE <date>}|{TIME <time>}|{TIMESTAMP <datetime>}
```

```
<temporal argument>::  
{DATE <date>}|{TIME <time>}|{TIMESTAMP <datetime>}|  
{INTERVAL <interval>}
```

```
<interval>::  
'<integer value>' {YEAR|MONTH|DAY|HOUR|MINUTE|SECOND}
```

使用此判式可测试两个年代的期间是否重叠。

例如, 以下 OVERLAPS 判式求值为 TRUE:

```
{DATE '01-01-2000', INTERVAL '03' MONTHS} OVERLAPS  
{DATE '03-15-2000', INTERVAL '10' DAYS}
```

因为第二个日期范围 (03-15-2000-03-25-2000) 的一部分处于 (或重叠) 第一个日期范围 (01-01-2000-04-01-2000) 部分之内。

类似地, 下面的 OVERLAPS 判式求值为 FALSE:

```
{TIME '09:23:00', TIME '13:45:00'} OVERLAPS  
{TIME '14:00:00', TIME '14:25:00'}
```

因为部分第二个时间期处于 (或重叠) 第一个时间期之内。

许多 DBMS 产品不支持 OVERLAPS 判式。如果支持, 很可能在存储过程中使用, 接受数据类型为 CHARACTER 的日期、时间和时间间隔等参数。用于存储日期或时间间隔的 CHARACTER 型参数将出现在 OVERLAPS 判式中, 以代替本例中使用的实际值。现在要了解的重要事情是, 如果第二个时间区间落在第一个时间区间之内, 则 OVERLAPS 判式返回 TRUE, 既可以指定两个时间区间作为开始和结束的日期/时间, 也可以指定开始日期/时间和一个持续期 (间隔)。

技巧 199 理解 GROUP BY 子句和组合查询

与总计函数类似, GROUP BY 子句也总结数据。但是, 不是生成单一的总计结果行, GROUP BY 子句生成多个分类汇总——表中一组表行有一个。

例如, 如果想要了解在上一年间顾客购买的货物总量, 可在 SELECT 语句中使用 SUM() 总计函数来生成单一行的结果表:

```
SELECT SUM(invoice_total) AS 'Total Sales' FROM invoices  
WHERE invoice_date >= (GETDATE() - 365)
```

其中结果表为:

```
Total Sales
```

```
-----
```

```
47369
```

另一方面，如果想要各个顾客的购货总量的明细，可添加 GROUP BY 子句，如下所示：

```
SELECT cust_ID, SUM(invoice_total) AS 'Total Sales'
FROM invoices
WHERE invoice_date >= (GETDATE() - 365)
GROUP BY cust_ID
```

此查询告诉 DBMS 生成每个顾客的购货总量的分类汇总如下：

```
cust_ID Total Sales
```

```
-----
```

```
1 7378
```

```
5 7378
```

```
7 22654
```

```
8 1290
```

```
9 8669
```

包括 GROUP BY 子句的查询（如本例所示）就称为组合查询，因为 DBMS 总会（总计）了源表中所选的行作为每一组中一行值。在 GROUP BY 子句中指定的列（在本例中是 CUST_ID）称为组合列，因为 DBMS 使用这些列值推出源表中的哪些行属于中间表中的哪个组。

在 DBMS 将中间结果表排列成行组，在此行组中的每行上的组合列都有相同的值，系统对组中的所有行计算总计函数（在 SELECT 子句中列出的）的值。最后，总计函数的结果与列在 SELECT 子句中的其他项目一起作为代表每组的一行都添加到最后的结果表中。

最后要了解的是，组合查询是受到某些限制的，对 GROUP BY 子句中列出的列以及在 SELECT 子句中列出的输出值表达式都有限制。

所有的组合列（GROUP BY 子句中列出的列）必须是来自 FROM 子句列出的表。由此，不能根据实际值、总计函数结果或任何其他表达式计算的值来对行分组。

技巧 200 使用 GROUP BY 子句根据单一列值组合行

正如读者在技巧 63 “使用 SELECT 语句从一个或多个表的行中显示列”中所学习的，SELECT 语句允许显示表中满足查询的 WHERE 子句中指定的搜索标准的所有行（如果没有 WHERE 子句，SELECT 语句将显示表中所有行的所有列中的值）。为了把由 SELECT 语句返回的行分成组并对每一组只显示一行的数据值，可向 SELECT 语句中添加 GROUP BY 子句，而执行组合查询。

在执行组合查询时，DBMS 执行以下步骤：

1. 根据列在查询的 FROM 子句中的表的笛卡尔积生成中间表。
2. 应用 WHERE 子句（如果有的话）中的搜索标准，从中间表（步骤 1 中产生的）中清除那些 WHERE 子句求值为 FALSE 的行。
3. 将中间表中余下的行排列成组，使得组中的每行上的组合列（列在 GROUP BY 子句中的）的值都相等。
4. 对于每个行组计算 SELECT 子句中的每个数据项的值并为每组生成查询结果中的一行。
5. 如果查询包括 HAVING 子句，则将搜索条件应用于结果表中的行并消除那些 HAVING

子句求值为 FALSE 的总计行。

6. 如果 SELECT 语句包括 DISTINCT 子句（读者已在技巧 169 “使用 DISTINCT 子句消除行集中的重复”中学习了有关知识），从结果表中清除所有重复的行。

7. 如果有 ORDER BY 子句，根据 ORDER BY 子句中列出的列对结果表中余下的行排序（读者在技巧 67 “使用 ORDER BY 子句指定由 SELECT 语句返回行的顺序”中学习了 ORDER BY 子句）。

例如，当执行如下的组合查询时：

```
SELECT state, COUNT(*) AS 'Customer Count' FROM customers
GROUP BY state
```

DBMS 将生成表明每个州中的顾客数的结果表。由于 FROM 子句只有一个表（CUSTOMERS），步骤 1 生成的中间表就由 CUSTOMERS 表中的所有行组成。因为没有 WHERE 子句，所以 DBMS 不会从中间表中取消行。在步骤 3 中，DBMS 将把中间表中的行排列成组，每组中的组合列（STATE）值在组中的每行上都相同。

接着，DBMS 对每组应用 COUNT(*) 函数，以便为表中的每组生成包括州代码和组中顾客计数的结果表行。由于既没有 HAVING 也没有 DISTINCT 子句，DBMS 不会从结果表中消除任何行，结果表如图 11.2 中 MS-SQL Server 应用程序窗口的下窗格中所示。

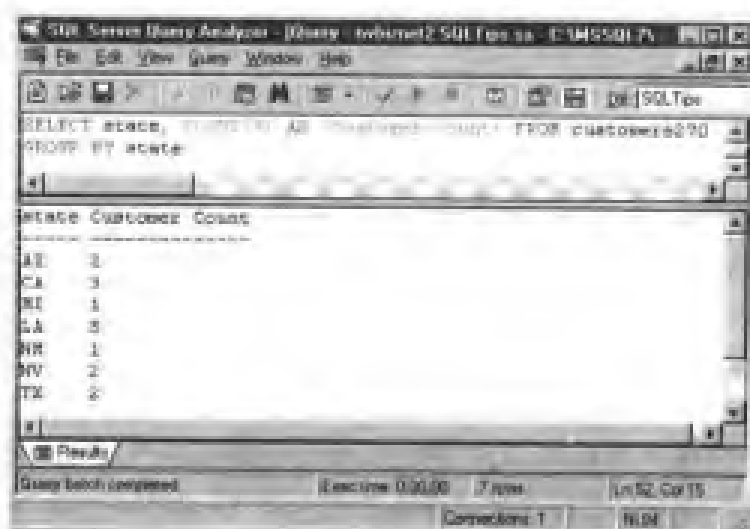


图 11.2 MS-SQL Server Query Analyzer 对单列组合查询的查询和结果

注意：由于没有 ORDER BY 子句，结果表中的行以组合列（STATE）的升序排列是巧合的。DBMS 将以排列中间表中的组的顺序来排列结果表中的行。因而，如果想让 DBMS 对结果表中的行根据一列或多列中的值以升序或降序排列，一定要包括 ORDER BY 子句。

技巧 201 使用 GROUP BY 子句根据多列组合行

在技巧 200 “使用 GROUP BY 子句根据单一列值组合行”中，读者学习了如何使用 GROUP BY 子句生成根据单一列值组合的源表行而得出总计（分类汇总）行的结果表。SELECT 语句中的 GROUP BY 子句中只有一列是组合查询的最简单形式。如果表中的行组依赖于多列，只要在查询的 GROUP BY 子句中列出定义组所需的所有列即可。在 SELECT 语句的 GROUP BY 子句中列出的列的数目没有上限，对组合列的惟一限制是，组合列必须是查询的 FROM 子句

中列出的表之一中的列。但是，请记住，无论在 GROUP BY 子句中列出多少列，标准的 SQL 将在查询的结果表中只显示组的分类汇总的一级。

例如，在技巧 200 中，读者了解到，可使用以下查询显示在每州中顾客数的结果表：

```
SELECT state, COUNT(*) AS 'Customer Count' FROM customers
GROUP BY state
```

如果想要各州内每位销售人员的该州顾客数的明细表，可执行以下查询：

```
SELECT state, salesperson, COUNT(*) AS 'Customer Count'
FROM customers
GROUP BY state, salesperson
```

此查询产生以下结果表：

state	salesperson	Customer Count
AZ	101	1
CA	101	3
LA	101	2
HI	102	1
LA	102	2
NV	102	2
TX	102	1
AZ	103	1
LA	103	1
NM	103	1
TX	103	1

此表根据 SALESPERSON 在 STATE 内组合了顾客数。但请注意，新查询只生成了根据每个（STATE、SALESPERSON）的分类汇总。标准 SQL 将不会在同一结果表中既给出根据 SALESPERSON 的分类汇总又给出根据 STATE 的分类汇总，即使在 GROUP BY 子句中列出这两列来也无济于事。

注意：由于标准 SQL 只为每个惟一的组合列（即 GROUP BY 子句中列出的）的组合给出一级分类汇总，因而必须使用编程的 SQL 将结果表传递给能够产生逐级分类汇总的应用程序。另一种选择是使用 ORDER BY 子句改变结果表中行的顺序（此内容将在技巧 202“使用 ORDER BY 子句改变由 GROUP BY 子句返回的组中的行序”中学习）。虽然 ORDER BY 子句本身并不产生任何分类汇总，但确实使得手工计算第二组分类汇总变得容易。只要用相同列值将行组合在一起即可。最后一种从单条 SQL 语句中直接获得多级分类汇总的方法是使用 MS-SQL Server Transact-SQL 的 COMPUTE 子句（此内容将在技巧 203“使用 MS-SQL Transact-SQL 的 COMPUTE 子句在同一结果表中显示明细及汇总行”中学习）。遗憾的是，COMPUTE 子句不是 SQL-92 标准的一部分，只能在 MS-SQL Server 上使用。

技巧 202 使用 ORDER BY 子句改变由 GROUP BY 子句返回的组中的行序

在技巧 67“使用 ORDER BY 子句指定由 SELECT 语句返回行的顺序”中，读者学习了如何使用 ORDER BY 子句对由非组合查询返回的结果表行排序。组合查询中的 ORDER BY 子句与在非组合查询中的使用方法是类似的。例如，为了对以下组合查询的结果表以每州中顾客数的降序排序：

```
SELECT state, COUNT(*) AS 'Customer Count' FROM customers
GROUP BY state
```

可改写 SELECT 语句以包括 ORDER BY 子句:

```
SELECT state, COUNT(*) AS 'Customer Count' FROM customers
GROUP BY state
ORDER BY "Customer Count" DESC
```

请注意,只限于根据列在 GROUP BY 子句中的列来对查询的结果表排序。与所有查询的情况一样,列在 ORDER BY 子句中的列只限于查询的 SELECT 子句中指定的列或列标题。因而,下面的每一条 ORDER BY 子句对于本例中 SELECT 语句来说都是合法的:

```
ORDER BY state
ORDER BY state "Customer Count"
ORDER BY "Customer Count" state
ORDER BY "Customer Count"
```

正如在技巧 201 “使用 GROUP BY 子句根据多列组合行”中所提到的,可以使用 ORDER BY 子句使得当复核由多个组合列产生的组合查询的结果表时手工计算第 2 级(或第 3 级、第 4 级等)分类汇总变得容易一些。例如,技巧 201 中的根据 (STATE、SALESPERSON) 组合查询的结果表中行的排列就使得手工分类汇总每个销售人员的顾客数变得容易,甚至通过查询只提供对每个 (STATE、SALESPERSON) 对的分类汇总时也是如此。只要在每个 SALESPERSON 变化时画一条横线并添加在行块(组)上添加 CUSTOMER COUNT 值。

作为对照,如果要根据州来计算顾客数的分类汇总,其任务要困难一些,因为相同的州缩写名并未在结果表中组合在一起。但是,如果改变组合查询中的 ORDER BY 子句如下:

```
SELECT state, salesperson, COUNT(*) AS 'Customer Count'
FROM customers
GROUP BY state, salesperson
ORDER BY state, "Customer Count"
```

就可生成下面的结果表:

state	salesperson	Customer Count
AZ	101	1
AZ	103	1
CA	101	3
HI	102	1
LA	101	2
LA	102	2
LA	103	1
NM	103	1
NV	102	2
TX	102	1
TX	103	1

这就使得根据对相同州代码列出的分组顾客数手工计算州顾客数变得容易。

技巧 203 使用 MS-SQL Transact-SQL 的 COMPUTE 子句在同一结果表中显示明细及汇总行

MS-SQL Server Transact-SQL 的 COMPUTE 子句允许对结果表中的行执行总计(列)函数

(SUM()、AVG()、MIN()、MAX()、COUNT())。因而，可在 SELECT 语句中使用 COMPUTE 子句生成既有明细又有总结信息的结果表。

例如，以下 SELECT 语句中的 COMPUTE 子句：

```
SELECT * FROM customers WHERE state IN ('CA','NV')
COMPUTE SUM(total_purchases), AVG(total_purchases),
COUNT(cust_ID)
```

将生成以下结果表：

cust_id	cust_name	state	salesperson	total_purchases
1	CA Customer 1	CA	101	78252.0000
2	CA Customer 2	CA	101	45852.0000
6	NV Customer 1	NV	102	12589.0000
7	CA Customer 3	CA	101	75489.0000
12	NV Customer 2	NV	102	56789.0000

sum

268971.0000

avg

53794.2000

cnt

5

此表中有有关公司的 California 和 Nevada 州的顾客的明细信息的行，并在报告中与总体总计和平均销售一起以表明顾客数的总计行结束。

注意：严格地说，带 COMPUTE 子句的 SELECT 违反了关系查询的基本原则，因为其结果不是表。由于 MS-SQL Sever 为在 COMPUTE 子句中的每个总计函数添加了两个标题行和总计行，查询返回不同类型行的组合。

虽然此子句对于行计数和总结结果表中的数值特别有用，但在 SELECT 语句中使用 COMPUTE BY 受到以下规则的限制：

- 只有来自 SELECT 子句中的行才能用于 COMPUTE 子句中。
- COMPUTE 子句中的总计函数不能约束为 DISTINCT。
- COMPUTE 子句不能用在 SELECT INTO 语句中。
- 在 COMPUTE 子句中只允许使用列名（而不是列标题）。

技巧 204 使用 MS-SQL Transact-SQL 的 COMPUTE 和 COMPUTE BY 子句显示多级分类汇总

在技巧 201 “使用 GROUP BY 子句根据多列组合行”中，读者学习了如何使用组合查询（带 GROUP BY 子句的 SELECT 语句）来组合源表数据并将其作为每组一个总计行显示在结果表中。从中还可以看出，组合查询只能显示单级分类汇总。因而，不能使用标准的组合查询在同一结果表中既显示组的分类汇总又显示总的汇总。但是，如果在单个非组合查询中使用

COMPUTE BY 和 COMPUTE 子句, 可生成既有分类汇总又有总汇总的结果表 (换句话说, 通过向没有 GROUP BY 子句的 SELECT 语句中添加 COMPUTE BY 子句和 COMPUTE 子句可生成多级分类汇总)。

例如, 以下查询中的 COMPUTE BY 和 COMPUTE 子句:

```
SELECT state, salesperson, total_purchases FROM customers
WHERE state IN ('LA','CA')
ORDER BY state, salesperson
COMPUTE SUM(total_purchases) BY state, salesperson
COMPUTE SUM(total_purchases)
```

将在下面的结果表中显示分类汇总和总汇总:

state	salesperson	total_purchases
CA	101	78252.0000
CA	101	45852.0000
CA	101	75489.0000
	sum	
		199593.0000
LA	101	74815.0000
LA	101	15823.0000
	sum	
		90638.0000
LA	102	96385.0000
LA	102	85247.0000
	sum	
		181632.0000
LA	103	45612.0000
	sum	
		45612.0000
	sum	
		517475.0000

除了 COMPUTE 子句的限制 (读者在技巧 203 “使用 MS-SQL Transact-SQL 的 COMPUTE 子句在同一结果表中显示明细及汇总行” 中学习的) 之外, 带 COMPUTE BY 子句的查询还必须遵守以下规则:

- 为了包括 COMPUTE BY 子句, SELECT 语句必须有 ORDER BY 子句。
- 列在 COMPUTE BY 子句中的列既可与列在 ORDER BY 子句中的列一样, 也可能是其子集。另外, 在两个子句 (ORDER BY 和 COMPUTE BY) 中的列必须以相同的顺序, 从左向右, 必须以相同的列开始, 而且不能跳过任何列。
- COMPUTE BY 子句不能包括任何标题名——只能有列名。

最后的“无标题”限制似乎暗含着不可能在以下的组合查询中执行使用 COMPUTE BY 子

句的查询来总计来自总计函数的分类汇总：

```
SELECT state, salesperson,  
SUM(total_purchases) AS 'Tot_Purchases'  
FROM customers  
GROUP BY state, salesperson  
ORDER BY state, salesperson
```

毕竟，结果表中的 TOT_PURCHASES 是标题而不是列名。因而，带每个 (STATE, SALESPERSON) 对的购货的分类汇总的列用在 COMPUTE BY 子句中是不合格的。

但是，如果执行如下的 CREATE VIEW 语句：

```
CREATE VIEW vw_state_emp_tot_purchases AS  
SELECT state, salesperson,  
SUM(total_purchases) AS 'Tot_Purchases'
```

此语句创建使用总计函数的标题（在本例中是 TOT_PURCHASES）作为列的虚拟表，就可使用 COMPUTE BY 子句对总计列的分类汇总进行分类汇总。在本例中的 VW_STATE_EMP_TOT_PURCHASES 视图中，TOT_PURCHASES 是列。因而，如果引用视图作为 SELECT 语句的 FROM 子句中的（虚拟）表，则可在如下的查询中使用 COMPUTE BY 子句：

```
SELECT state, salesperson, tot_purchases  
FROM vw_state_emp_tot_purchases  
ORDER BY state, salesperson  
COMPUTE SUM(tot_purchases) BY state
```

从而在同一结果表中，既显示每个 SALESPERSON 按 STATE（总计来自组合查询的分类汇总）的分类销售额以及总计的销售额，又可显示根据 STATE 的销售额。

技巧 205 理解 GROUP BY 子句如何看待 NULL 值

当 NULL 值出现在组合查询的一个（或多个）组合列中时由 NULL 值产生的问题类似于 NULL 值对总计函数和搜索条件产生的问题。由于组定义为组合列的复合值相等的行组成的行集，那么当定义组的组合列的一个或多个列中的值为未知（NULL）时，什么组应该包括这样的行呢？

如果 DBMS 遵循 WHERE 子句中搜索条件所用的规则，那么 GROUP BY 对于在组合列中的任何一列中带有 NULL 值的每一行都应该自成一组。毕竟，在 WHERE 子句中两个 NULL 值的相等测试的结果总为 FALSE，因为根据 SQL 标准，NULL 不等于 NULL。因而，如果一行有带有 NULL 值的组合列，它就不能放入另一带有 NULL 值的组合列的行的组中，因为同一组的所有行必须有相匹配的组合列值（而 NULL \neq NULL）。

由于 SQL 的设计人员看到为每一个在组合列中带有 NULL 值的行创建一个独立的组既混乱又无用，所以他们编写 SQL 标准时使得在 GROUP BY 子句的目的下将 NULL 值看作是相等的。因而，如果两行在相同的组合列中有 NULL 值，而在其余的非 NULL 的组合列中的值相匹配，DBMS 将把这些行组合在一起。

例如，以下组合查询：

```
SELECT state, salesperson,  
SUM(amount_purchased) AS 'Total Purchases'  
FROM customers  
GROUP BY state, salesperson
```

ORDER BY state, salesperson

对于包括以下行的 CUSTOMERS 表:

state salesperson amount_purchased

state	salesperson	amount_purchased
NULL	NULL	45612.0000
NULL	NULL	15826.0000
NULL	101	45852.0000
NULL	101	74815.0000
NULL	101	75489.0000
AZ	NULL	75815.0000
AZ	103	36958.0000
CA	101	78252.0000
LA	NULL	96385.0000
LA	NULL	85247.0000

将显示下面的结果表:

state salesperson Total Purchases

state	salesperson	Total Purchases
NULL	NULL	61438.0000
NULL	101	196156.0000
AZ	NULL	75815.0000
AZ	103	36958.0000
CA	101	78252.0000
LA	NULL	181632.0000

技巧 206 使用 HAVING 子句筛选包括在组合查询结果表中的行

在 WHERE 子句去除了不想要的行之后, DBMS 使用 HAVING 子句作为筛选器删除那些总计或单独列值不能满足 HAVING 子句中的搜索条件的行组 (而不是单独的行)。

例如, 在以下查询中的 WHERE 子句:

```
SELECT RTRIM(first_name)+' '+last_name AS 'Employee Name',
SUM(amt_purchased) AS 'Total Sales'
FROM customers, employees
WHERE customers.salesperson = employees.emp_ID
GROUP BY RTRIM(first_name)+' '+last_name
HAVING SUM(amt_purchased) > 250000
ORDER BY "Total Sales"
```

告诉 DBMS 一次一行地检查由 CUSTOMERS 和 EMPLOYEES 表组成的中间表中的行并清除那些在 EMP_ID 中的值不等于 SALESPERSON 列中的值的行。接着, DBMS 根据雇员名 (由 GROUP BY 子句指定的) 组合其余行。然后 DBMS 使用 HAVING 子句中的搜索标准来检查每组中的行。在本例中, 系统计算每个行组中的 AMT_PURCHASED 列的和, 并清除那些总计函数 (SUM(AMT_PURCHASED)) 返回等于或小于 250,000 的那些组中的行。

虽然 HAVING 子句中的搜索条件可测试单独的列值以及总计函数的结果, 但把单独列值测试放在查询的 WHERE 子句中 (而不是 HAVING 子句中) 会更有效率。例如, 以下查询:

```
SELECT emp_ID, RTRIM(first_name)+' '+last_name
AS 'Employee Name', SUM(amt_purchased) AS 'Total Sales'
```

```
FROM customers, employees
WHERE customers.salesperson = employees.emp_ID
GROUP BY RTRIM(first_name)+' '+last_name
HAVING (SUM(amt_purchased) < 250000) AND (emp_ID >= 102)
ORDER BY "Total Sales"
```

将在 HAVING 子句中测试 EMP_ID 列的值以便在最后的結果表中清除那些总銷售等于或超过 250,000 且 EMP_ID 小于 102 的雇员，这没有下面的查询更有效率：

```
SELECT emp_ID, RTRIM(first_name)+' '+last_name
       AS 'Employee Name', SUM(amt_purchased) AS 'Total Sales'
FROM customers, employees
WHERE (customers.salesperson = employees.emp_ID)
AND (emp_ID >= 102)
GROUP BY RTRIM(first_name)+' '+last_name
HAVING (SUM(amt_purchased) < 250000)
ORDER BY "Total Sales"
```

这个查询在 WHERE 子句中测试 EMP_ID 值（为了产生相同的结果表）。

在第一条查询中（在 HAVING 子句中测试 EMP_ID 列），DBMS 将对几个雇员组（那些 EMP_ID 值处于 001-101 范围内的）计算 AMT_PURCHASED 列的和，当系统检查组的 EMP_ID 值时，以清除那些组中的那些行。在第二条查询中，通过从中间结果表中清除行避免了将 EMP_ID 小于 102 的行排列成组并为这些组计算其总销售额的操作，然后 DBMS 将行排列成组并对每组应用 HAVING 子句中的总计函数（SUM()），这样就更有效率。

技巧 207 理解在组合查询中使用 HAVING 子句的 SQL 规则

正如读者在技巧 206 “使用 HAVING 子句筛选包括在组合查询结果表中的行”中所学习的，可使用 WHERE 子句或 HAVING 子句从查询结果中排除行或将行包括在查询结果中。由于 DBMS 使用 WHERE 子句中的搜索标准一次只过滤一行，WHERE 子句中的表达式必须对单独的行是可计算的。而在 HAVING 子句的搜索条件中的表达式对一个行组必须求值为单一值。因而 WHERE 子句中的搜索条件由使用列引用和实际值（常数）的表达式组成。另一方面 HAVING 子句的搜索条件通常由带有一个或多个总计（列）函数（如 COUNT()、COUNT(*)、MIN()、MAX()、AVG()或 SUM()）组成。

当执行带 HAVING 子句的组合查询时，DBMS 执行以下步骤：

1. 根据列在 SELECT 语句的 FROM 子句中的表生成的笛卡尔积创建中间表。如果 FROM 子句中只有一个表，那么中间表就是源表的副本。
2. 如果有 WHERE 子句，将其搜索条件应用于从中间表中过滤出不想要的行。
3. 将中间表中的行排列成行组，其中所有的组合列都有相同的值。
4. 将 HAVING 子句中的每个搜索条件应用于每个行组。如果行组不满足一个或多个搜索条件，则从中间表中删除该行组。
5. 计算查询的 SELECT 子句中的每项的值并为每个行组生成单一（总计）行。如果 SELECT 子句中的项目引用了一列，则在总计行使用来自组内所有行上的该列值。如果 SELECT 子句中的项目是个总计函数，则对要总计的行组计算函数值并将此值添加到该组的总计行。
6. 如果查询包括关键词 DISTINCT（如 SELECT DISTINCT），则从结果表中清除任何重

复的行。

7. 如果有 ORDER BY 子句, 则根据列在 ORDER BY 子句中的列值对结果表排序。

技巧 208 理解 SQL 如何处理 HAVING 子句的 NULL 结果值

HAVING 子句类似于 WHERE 子句, 可有 3 个值之一, 即 TRUE、FALSE 或 NULL。如果 HAVING 子句对行组求值为 TRUE, DBMS 使用组中行的值来生成结果表中的总计行。作为对照, 如果 HAVING 子句对行组求值为 FALSE 或 NULL, DBMS 在结果表中不总计该组中行的值。这样一来, DBMS 处理求值为 NULL 的 HAVING 子句的方式与处理求值为 NULL 的 WHERE 子句的方式一样, 在结果表中忽略产生 NULL 值的行。

请记住, 组列中的 NULL 值并不总引起 HAVING 子句中的搜索条件求值为 NULL。例如, 以下的查询:

```
SELECT state, COUNT(*) AS 'Customer Count',
       (COUNT(*) - COUNT(amt_purchased)) AS 'NULL Sales Count',
       SUM(amt_purchased) AS 'Sales' FROM Customers
GROUP BY state
HAVING SUM(amt_purchased) < 50000
```

产生的结果表类似于图 11.3 所示的 MS-SQL Server Query Analyzer 的下半部的结果窗格中的结果表, 即使在 California 顾客行的组中四个 AMT_PURCHASED 值中的 3 个都是 NULL。

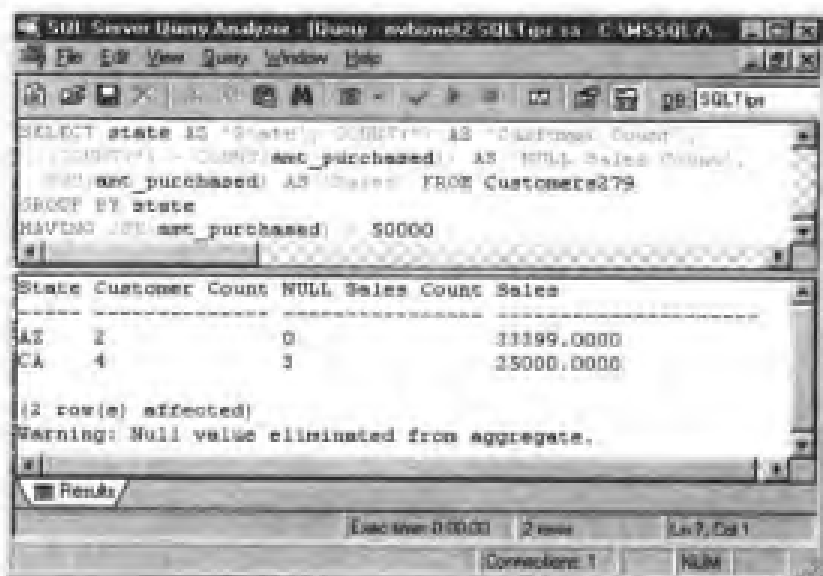


图 11.3 MS-SQL Server Query Analyzer 对带 HAVING 子句的组合查询的查询和结果

正如读者在技巧 86 “使用 SUM() 总计函数计算列值的总和” 中所学习的, 当对列值求和时, SUM() 总计函数忽略 NULL 值。因而, 在本例中, HAVING 子句中的搜索条件对 California 的顾客求值为 TRUE, 因为 SUM() 总计函数忽略了组中的 3 个 NULL 值的 AMT_PURCHASED 列值, 并返回非 NULL 的总计结果 (25,000)。

另一方面, 如果组中所有行的 AMT_PURCHASED 列中的所有值都为 NULL (比如对 California 的顾客), SUM() 函数将返回 NULL。结果, HAVING 子句将求值为 NULL, 因而 SELECT 语句将不在结果表中总计该组行。对本例来说, 如果所有 California 的顾客行在

AMT_PURCHASED 列上都有 NULL 值，SELECT 语句将产生如下结果表：

```
State Customer Count NULL Sales Count Sales
```

```
-----
```

State	Customer Count	NULL Sales Count	Sales
AZ	2	0	33399.0000

其中没有提到 California 顾客。

但是，如果将示例的查询改变一下，添加 IS NULL 搜索条件，如下所示：

```
SELECT state, COUNT(*) AS 'Customer Count',
  (COUNT(*) - COUNT(amt_purchased)) AS 'NULL Sales Count'
SUM(amt_purchased) AS 'Sales' FROM Customers
GROUP BY state
HAVING (SUM(amt_purchased) < 50000)
OR (SUM(amt_purchased) IS NULL)
```

HAVING 子句将对 California 顾客求值为 TRUE（因为 SUM() 总计函数返回 NULL 值），

而 SELECT 语句将显示结果表如下：

```
State Customer Count NULL Sales Count Sales
```

```
-----
```

State	Customer Count	NULL Sales Count	Sales
AZ	2	0	33399.0000
CA	4	4	NULL

现在要理解的重要事情是，对 HAVING 子句求值为 NULL（或 FALSE）的任何行组都将从组合查询的结果表中忽略。

第 12 章 使用 SQL 的联合语句和其他多表查询

技巧 209 使用来自多个 MS-SQL Server 数据库中的表

在所有以前技巧的示例中 SELECT 语句引用单一数据库 (SQLTips) 中的表, MS-SQL Server 允许用户执行使用来自多个数据库的表的 SQL 语句。

如果只使用名称引用表, DBMS 假设想要使用的表是当前数据库自己拥有的表。例如, 如果作为用户 FRANK 登录到 SQLTips 数据库并执行以下查询:

```
SELECT * FROM customers
```

如果 CUSTOMERS 表存在于 SQLTips 数据库中而且为 FRANK 所拥有, DBMS 将显示 CUSTOMERS 表的内容。

如果想要使用存在于当前数据库但为不同的用户名所拥有的表, 则表引用必须使用以下包括表的拥有者名的句法:

```
<owner name>.<table name>
```

由此, 如果以用户 FRANK 的身份登录到 SQLTips 数据库, 而且想要查询 MARY 所拥有的 CUSTOMERS 表, 可执行以下 SELECT 语句:

```
SELECT * FROM mary.customers
```

要使用另一数据库中的表, 表引用必须使用以下句法指定表所在数据库以及表的拥有者和表的名称:

```
<database name>.<owner name>.<table name>
```

例如, 如果登录到 SQLTips 数据库并想要查询 PUBS 数据库中为 DBO 所拥有的 AUTHORS 表, 可执行类似于下面的 SELECT 语句:

```
SELECT * FROM pubs.dbo.authors
```

正如在本技巧的开始时所提到的, 在同一 SQL 语句中可使用来自多个数据库中的表——只要使用必需的句法, 告诉 DBMS 到哪里可以找到想要使用的表。例如, 如果想要交叉引用当前数据库 (SQLTips) 中的 EMPLOYEES 表以及 PUBS 数据库中的 AUTHORS 表, 以便查看是否为自己工作的出版物作者, 可执行类似于下面的 SELECT 语句:

```
SELECT emp_ID, au_lname, au_fname  
FROM employees, pubs.dbo.authors  
WHERE employees.SSAN = pubs.dbo.authors.au_ID
```

由于查询在其引用 CUSTOMERS 表时忽略了数据库名和拥有者名, DBMS 就认为该表存在于当前数据库 (SQLTips) 中。

相对的, 由于查询引用 AUTHORS 表时包括了数据库名和拥有者名, DBMS 就知道使用 PUBS 数据库中为 DBO 所拥有的 AUTHORS 表。

注意: 当然 DBMS 将检查其系统表, 以保证用户名所指用户具有执行所提交的 SQL 语句的访问权限。如果没有此权限, DBMS 将中止语句的执行并显示以下错误信息:

```
Server: Msg 229, Level 14, State 5, Line 1
```

```
SELECT permission denied on object 'authors', database  
'pubs', owner 'dbo'.
```

简短地说，DBMS 不允许使用完全合格的表名（<database name>.<owner name>.<table name>）来绕过系统安全检查。

技巧 210 使用 FROM 子句执行多表查询

顾名思义，FROM 子句列出 DBMS 从中获得用于查询的数据值的表。例如，如下所示在 FROM 子句中带有单表的查询：

```
SELECT * FROM customers
```

是容易理解的——从 CUSTOMERS 表中提取每一行并向查询的 SELECT 子句中的“显示所有列值”（星号的意义）表达式提供其列值。

如果查询需要来自多个表中的数据，FROM 子句（总是列出查询数据的所有源表）将不止有一个的表。但是，不管 FROM 子句列出的是单表还是多个表，SELECT 语句中的各种子句总是一次处理一行数据。因而，如果 FROM 子句有多个表，DBMS 将计算这些表的笛卡尔积以创建可从中一次提取一行的单一表。

中间表（笛卡尔积）由 FROM 子句中列出的所有表中所有行的所有可能组合组成。每行由 FROM 子句中列出的所有表中的所有列组成。

例如，假设有一个 12,000 行的 STUDENTS 表，每行有 15 列（属性），还有一个有 1,000 行的 CLASSES 表，每行有 5 列。当执行以下查询时：

```
SELECT * FROM students, classes
```

DBMS 要创建有 12,000,000 行（12,000×1,000）和每行 20 列（15 列来自 STUDENTS 表再加上 5 列来自 CLASSES 表）的一个虚拟表。如果再有第 3 个表，如 TEACHERS 表，其中有 1,000 行，每行 10 列，则 DBMS 将在以下查询的 FROM 子句中组合 3 个表：

```
SELECT * FROM students, classes, teachers
```

此查询产生的中间虚拟表有 12,000,000,000 行（12,000×1,000×1,000），每行有 30 列（15 + 5 + 10）。正如读者所看见的，多表 SELECT 语句的笛卡尔积变得非常大。

当然，没有一种 DBMS 产品真正地上创建 FROM 子句列出的表的笛卡尔积的物理表。但是，涉及大表的多表查询有时要运行几个小时，因为 DBMS 在内存中一次一个复合行地创建并存储处理来自多表的数据所需的部分虚拟的中间表。

要理解的重要事情是，多表查询实际上使用的是单个（虽然可能很大）虚拟表，是 DBMS 根据 SELECT 语句的 FROM 子句中列出的表生成的笛卡尔积所创建的。这个概念是十分重要的，因为它有助于解释为什么单表查询需要 WHERE 子句过滤掉不想要的行。

技巧 211 使用 WHERE 子句联合与单列 PRIMARY KEY/FOREIGN KEY 对相关的两个表

当执行多表查询时，DBMS 把一个表中的行与另一表中的行联合起来创建一个中间的虚拟表，其中有来自两个源表中的所有数据。然后系统使用 SELECT 语句的子句来过滤并显示中间（联合的）表中的数据值。

最简单的多表查询形式是基于两个表中的行对之间父/子关系的等价联合（或自然联合）。DBMS 通过把父表中的 PRIMARY KEY 列值与子表中的 FOREIGN KEY 列值匹配将父表中的

行与子表中的行联合起来。图 12.1 显示的是由父表中的 PRIMARY KEY 列与子表的 FOREIGN KEY 列定义的 CUSTOMERS（父）表与 INVOICES（子）表之间的关系。

CUSTOMERS（顾客）表

ID	F_NAME	L_NAME	ADDRESS
1001	Konrad	King	765 E. Eldorado Lane
1003	Walter	Winchell	214 State Street
1009	Sally	Springer	77 Sunset Strip
⋮	⋮	⋮	⋮

主键

INVOICES（发票）表

INV_NUM	CID	INV_DATE	SHIP_DATE	INV_TOTAL	AMT_PAID
2001	1003	01/01/2000	02/05/2000	4500	4500
⋮	⋮	⋮	⋮	⋮	⋮
3010	1001	07/01/2000	08/15/2000	17500	14300
⋮	⋮	⋮	⋮	⋮	⋮
2730	1003	05/01/2000	07/05/2000	23750	0
⋮	⋮	⋮	⋮	⋮	⋮
9050	1003	09/29/2000	NULL	19300	0
⋮	⋮	⋮	⋮	⋮	⋮

外键

图 12.1 以共同列——CUSTOMERS 表中的 ID 列和 INVOICES 表中的 INV_NUM 列相关的 CUSTOMERS（父）表和 INVOICES（子）表

为了显示发票及下了每一定单的顾客的清单，DBMS 必须把每个 INVOICES 表中的子行与其 CUSTOMERS 表中的父行联合起来。例如，类似下面的查询将执行 CUSTOMERS 和 INVOICES 表间的等价联合：

```
SELECT f_name, l_name, inv_num, inv_date,
       (inv_total - amt_paid) AS 'Balance Due'
FROM invoices, customers
WHERE CID = ID
```

该查询的 FROM 子句告诉 DBMS 通过把 INVOICES 表中的行与 CUSTOMERS 表中的行联合起来创建中间虚拟表。接着系统使用 WHERE 子句中的搜索条件筛选出不想要的中间表行。在涉及父子表的等价联合中，DBMS 只把那些 FOREIGN KEY 列与 PRIMARY KEY 列匹配的中间表行传递到 SELECT 子句中。在本例中的等价联合查询将生成下面的结果表：

f_name	l_name	inv_num	inv_date	Balance Due
Walter	Winchell	2001	2000-01-01 00:00:00.000	.0000
Konrad	King	3010	2000-07-01 00:00:00.000	3200.0000

```
Walter Winchell 2730 2000-05-01 00:00:00.000 23750.0000
Walter Winchell 9050 2000-09-29 00:00:00.000 19300.0000
```

SQL 并不要求 SELECT 子句包括 WHERE 子句中所用的列。事实上，键数列（用于建立表间父/子关系）常常是 ID 号。虽然数字键值对于计算机惟一地识别并形成行对关系较为容易，但它们在结果表中没有什么意义。毕竟，使用查询结果的人（而不是机器）最可能要通过名称和描述而不是号码来了解和引用顾客、雇员、存货项目等。

技巧 212 使用 WHERE 子句联合与复合的 PRIMARY KEY/FOREIGN KEY 对相关的两个表

在技巧 130“理解引用完整性检查和外键”中，读者已经了解到，DBMS 仅当行的 FOREIGN KEY 列值与父表中已有行的 PRIMARY KEY 列值匹配时，才允许将该行插入子表中。但是正如读者在技巧 211“使用 WHERE 子句联合与单列 PRIMARY KEY/FOREIGN KEY 对相关的两个表”中所学习的，在涉及父/子表对的查询中，系统并不自动地使用匹配的 PRIMARY KEY/FOREIGN KEY 列值需求来过滤掉不想要的行。

为了使用两表间的父/子关系，必须执行一种等价联合查询。SELECT 语句的 WHERE 子句中的搜索条件告诉 DBMS 想要使用的只是联合后的行，其中来自父表行中的 PRIMARY KEY 值与来自子表行中的 FOREIGN KEY 值相匹配。

例如，如果有一个 INVENTORY（子）表，其中有单列 FOREIGN KEY，还有一个 ITEM_MASTER（父）表，其中有单列的 PRIMARY KEY，可使用以下 SELECT 语句：

```
SELECT inventory.item_number, description, qty_on_hand
FROM inventory, item_master
WHERE inventory.item_number = item_master.item_number
ORDER BY description
```

来显示货品描述（来自于 ITEM_MASTER 表）以及存货目录中该货品的数量，两者在结果表中作为单个的联合的行。

正如读者在技巧 127“理解单列和复合键字”中所学习的，有时有复合的键值，即表中的 PRIMARY KEY（或 FOREIGN KEY）值是由多列值组成的。例如，假设购买了不同卖主带有相同货品号的物品而且想要找出每个卖主的产品在手头的数量。如果使用由（ITEM_NUMBER、VENDOR_CODE）组成的复合 PRIMARY KEY，前述带有单个搜索条件的查询将生成下面的不正确结果：

item_number	description	qty_on_hand
1	Item 1 from Vendor 1	111
2	Item 2 from Vendor 1	222
2	Item 2 from Vendor 1	111
2	Item 2 from Vendor 1	111
2	Item 2 from Vendor 2	222
2	Item 2 from Vendor 2	111
2	Item 2 from Vendor 2	111
3	Item 3 from Vendor 1	333
3	Item 3 from Vendor 3	333

而实际上存货目录如下:

item_number	description	qty_on_hand
1	Item 1 from Vendor 1	111
2	Item 2 from Vendor 1	111
3	Item 3 from Vendor 1	111
2	Item 2 from Vendor 2	222
3	Item 3 from Vendor 3	333

为了执行使用由相关表中匹配的复合(多列) FOREIGN KEY/PRIMARY KEY 值定义的父/子关系的等价联合查询, SELECT 语句的 WHERE 子句必须有一条搜索条件来匹配每个组成复合键字的列对。例如,如果有两个与复合键字(如 ITEM_NUMBER、VENDOR_CODE)相关的表,执行以下查询:

```
SELECT inventory.item_number, description, qty_on_hand
FROM inventory, item_master
WHERE (inventory.item_number = item_master.item_number)
AND (inventory.vendor_code = item_master.vendor_code)
ORDER BY description
```

其中 WHERE 子句的搜索条件告诉 DBMS 过滤掉所有联合的行,但那些组成复合 FOREIGN KEY 值和 PRIMARY KEY 值具有匹配值的列对除外(如果每个复合键值由 3 个列组成, WHERE 子句中应该有由 AND 运算符连接的 3 个相等表达式 [每个匹配的列对一个])。

技巧 213 使用 WHERE 子句根据父/子关系联合 3 个或多个表

正如读者在技巧 211 “使用 WHERE 子句联合与单列 PRIMARY KEY/FOREIGN KEY 对相关的两个表”和技巧 212 “使用 WHERE 子句联合与复合的 PRIMARY KEY/FOREIGN KEY 对相关的两个表”中所学习的,每当系统执行两表查询时, DBMS 都生成由 SELECT 语句的 FROM 子句中列出的表形成的笛卡尔积。接着 DBMS 使用查询的 WHERE 子句中的一个或多个搜索条件过滤掉那些无关的行对。如果把查询中要联合的表从 2 增加到 3 (或更多), DBMS 仍然要进行同样的过程来生成这些表的笛卡尔积,然后从中间虚拟表(笛卡尔积)中过滤掉无关的行。

用来过滤掉无关的联合行的 WHERE 子句的每个搜索条件必须识别一对列(一系列来自每个相关表对),如果中间表中的联合行表达了合法的父/子关系,列对的值必须匹配。例如,在以下两表查询中的 WHERE 子句过滤掉与无关的顾客明细行联合的发票明细行(相反亦可):

```
SELECT f_name, l_name, inv_num, inv_date,
       (inv_total - amt_paid) AS 'Balance Due'
FROM invoices, customers
WHERE invoices.CID = customers.ID
```

两表间的父/子关系要求,子表的 FOREIGN KEY 值(INVOICES.CID)必须与父表的 PRIMARY KEY 值(CUSTOMERS.ID)相匹配。因而在本例中的 WHERE 子句删除了所有那些 PRIMARY KEY/FOREIGN KEY 列对值不匹配的联合行,因为这些行不能表达合法的两表间的父/子关系。

类似地,为了过滤掉对 3 个(或更多)表的查询中笛卡尔积中的“垃圾”行, WHERE 子句必须测试包括至少来自每个源表中的一列的列值对。例如,为了获得包括发票余额和销售人

员名称的顾客清单，可提交以下 3 表查询：

```
SELECT f_name, l_name, inv_no, inv_date,
       (inv_total - amt_paid) AS 'Balance Due',
       RTRIM(first_name)+' '+last_name AS 'Salesperson'
FROM invoices, customers, employees
WHERE (customers.ID = invoices.CID)
AND (invoices.salesrep = employees.emp_id)
```

WHERE 子句中的搜索条件过滤掉那些来自 CUSTOMERS（父）表中的 ID（PRIMARY KEY）与来自 INVOICES（子表）的 CID（FOREIGN KEY）不匹配的联合行，还要过滤掉那些来自 INVOICES（子）表中的 SALESREP（FOREIGN KEY）与来自 EMPLOYEES（父）表中的 EMP_ID（PRIMARY KEY）不匹配的联合行。

正如本例所表明的，对 3 表（或更多表）查询中的 WHERE 子句并不测试每个表的相同匹配的列对值。惟一的要求是，WHERE 子句必须检查至少一对来自每套父/子表（由查询的 FROM 子句列出）的 FOREIGN KEY/PRIMARY KEY 值。

技巧 214 使用 WHERE 子句根据非键字列联合表

在技巧 211~技巧 213 中的示例多表中的所有 WHERE 子句过滤掉那些来自子表的 FOREIGN KEY 值与来自父表的 PRIMARY KEY 值不匹配的联合行。但是，SQL 还允许根据非 PRIMARY KEY 和 FOREIGN KEY 列值的匹配来联合行。例如，假设公司对不同的部门有不同的一套培训手册。以下查询中的 WHERE 子句将过滤掉那些把一个部门的雇员与另一部门的手册联合起来的行：

```
SELECT first_name, last_name, title
FROM manuals, employees
WHERE manuals.for_dept = employees.dept
```

上面查询产生以下结果表：

first_name	last_name	title
Richard	Kimbal	Handling Complaints
Richard	Kimbal	Efficient Order Taking
Richard	Kimbal	Frequently Asked Questions
Hellen	Waters	Handling Complaints
Hellen	Waters	Efficient Order Taking
Hellen	Waters	Frequently Asked Questions
Ed	Norton	Prospecting
Ed	Norton	Working Callbacks
Ed	Norton	Making Referral Calls
Steve	Forbes	Prospecting
Steve	Forbes	Working Callbacks
Steve	Forbes	Making Referral Calls
Charles	Coulter	Mechanics of the Pre-Close
Charles	Coulter	Successful Closing Strategies
Charles	Coulter	Proper Menu Planning
Ralph	Cramden	Mechanics of the Pre-Close

Ralph	Cranden	Successful Closing Strategies
Ralph	Cranden	Proper Menu Planning

以上查询根据非键字列值的匹配生成结果表，其中有表明两表间多对多关系的联合行。例如，在本例的结果表表明，根据匹配的部门值，来自 MANUALS 表中的每行都是与来自 EMPLOYEES 表的几行相关的。另外，来自 EMPLOYEE 表的每行都有与来自 MANUALS 表中的几行相同的部门（因而也就相关了）。

另一方面，根据 PRIMARY KEY/FOREIGN KEY 值对的匹配来联合多个表可生成表明来自父表中的每行与子表中的子行（相关行）之间一对多关系的结果表。或者说，反向地看一下关系，结果表表明把键值与父表中单一（父）行与子行（子表中的行）匹配而相关的子行之间的多对一关系。

例如，假设有一个 EMPLOYEES 父表和 TIMECARDS 子表。有下面的 SELECT 语句：

```
SELECT first_name, last_name, card_date, start_time,
stop time
FROM employees, timecards
WHERE employees.emp_ID = timecards.emp_ID
```

将根据 PRIMARY KEY（EMPLOYEES.EMP_ID）和 FOREIGN KEY（TIMECARDS.EMP_ID）值的匹配对，把父行（来自 EMPLOYEES 表）与子行（来自 TIMECARDS 表）联合起来。在 WHERE 子句过滤掉联合的无关行之后，结果表显示来自 EMPLOYEES 表（父表）的每行与来自 TIMECARDS（子）表的 0 个或多个行的联合（通过匹配的键字列值对）。或者换一种说法，结果表将表明来自 TIMECARDS（子）表中的一行或多行与来自 EMPLOYEES（父）表中的一行且只有一行联合（通过匹配的键字列值对）。

在前面的多对一和一对多关系的讨论中要理解的重要事情是，不管是根据匹配的键字列值对还是根据匹配的非键字列值对来联合表，编写 SELECT 语句的 WHERE 的方式是一样的。在两种情况下，WHERE 子句包括一条比较测试可过滤掉那些两个表中的联合相关行的列对值不匹配的行。

技巧 215 理解非等价联合

虽然技巧 211~技巧 214 中所有的多表查询的示例都根据两表中共同的列对的相等性来联合表，SQL 还允许根据列对间的非相等关系来联合表。例如，假设想要生成雇员和根据雇用年限所应享受的福利的清单。给定 EMPLOYEES 表和无关系的 BENEFITS 表，可执行以下 SELECT 语句：

```
SELECT first_name, last_name,
CAST((GETDATE() - date_hired) AS INTEGER)
AS 'Days Employed',
description AS 'Eligible For'
FROM employees, benefits
WHERE CAST((GETDATE() - date_hired) AS INTEGER) >=
days_on_job_required
ORDER BY emp_ID
```

来生成如下的结果表：

first_name	last_name	Days Employed	Eligible For
------------	-----------	---------------	--------------

Robert	Cunningham	3440	Retirement Plan
Robert	Cunningham	3440	Paid Vacation
Robert	Cunningham	3440	Paid Sick Days
Robert	Cunningham	3440	Paid Dental
Robert	Cunningham	3440	Paid Medical
Lori	Swenson	153	Paid Dental
Lori	Swenson	153	Paid Medical
Richard	Kimbal	93	Paid Dental
Richard	Kimbal	93	Paid Medical
Glenda	Widmark	32	Paid Medical

结果表中的每行都把雇员姓名和雇用年限（来自 EMPLOYEES 表）与根据雇用年限应享受的福利（来自 BENEFITS 表）联合起来。

不管是根据等价联合还是非等价联合，WHERE 子句中的比较测试过滤掉那些用于联系两个表的列对值不满足定义表间关系的条件的联合行。

在等价联合中，WHERE 子句使用相等运算符来比较用于联系表的列对值并过滤掉那些列对值不等的联合行。类似地，在本例中显示的非等价联合中，WHERE 子句使用大于或等于(>=)运算符来过滤那些雇用天数表达式小于 DAYS_ON_JOB_REQUIRED 列（来自 BENEFITS 表）值的联合行。

要理解的重要事情是，当执行多表查询时，DBMS 总是生成列在 SELECT 语句的 FROM 子句中的表的笛卡尔积，然后使用 WHERE 子句中的一个或多个搜索条件过滤掉那些相关的（成对的）列中值不满足比较运算符（用于定义两表间关系）条件的行。

技巧 216 在有一个或多个相同列名的联合的多个表中多表查询中使用合格的列名

当编写多表查询时，如果想要获得数据的列名对在查询中联合的一个表中惟一时，在 SELECT 语句中可使用列名从一个表中提取数据。例如，如果有一个由以下 CREATE 语句定义的 CUSTOMERS 表和 EMPLOYEES 表：

```
CREATE TABLE customers      CREATE TABLE employees
(cust_ID      INTEGER,        (emp_ID      INTEGER,
cust_f_name   VARCHAR(30),    emp_f_name VARCHAR(30),
cust_l_name   VARCHAR(30),    emp_l_name VARCHAR(30))
salesperson   INTEGER)
```

可执行下面的 SELECT 语句通过列名来提取列数据：

```
SELECT RTRIM(cust_f_name)+' '+cust_l_name AS 'Customer',
RTRIM(emp_f_name)+' '+emp_l_name AS 'Salesperson'
FROM customers, employees
WHERE salesperson = emp_ID
```

DBMS 自动知道要提取来自 CUSTOMERS 表中的顾客的名和姓（CUST_F_NAME 和 CUST_L_NAME）以及来自 EMPLOYEES 表的雇员的名和姓（EMP_F_NAME 和 EMP_L_NAME）。毕竟 DBMS 可在一个且只有一个被查询的源表中找到查询中使用的每个列名。

另一方面，如果执行的多表查询中，所需数据的列名可在查询联合的不止一个表中找到，此时必须在 SELECT 语句中使用合格的列名。合格的列名告诉 DBMS 要提取数据的表名和列名。

例如, 如果 CUSTOMERS 和 EMPLOYEES 表是用以下 CREATE 语句创建的:

```
CREATE TABLE customers      CREATE TABLE employees
(cust_ID      INTEGER,      (emp_ID INTEGER,
cust_f_name VARCHAR(30),    f_name  VARCHAR(30),
cust_l_name VARCHAR(30),    l_name  VARCHAR(30))
emp_ID INTEGER)
```

DBMS 会中止以下 SELECT 语句的执行:

```
SELECT RTRIM(cust_f_name)+' '+cust_l_name AS 'Customer',
RTRIM(f_name)+' '+l_name AS 'Salesperson'
FROM customers, employees
WHERE emp_ID = emp_ID
```

并显示下面的错误信息:

```
Server: Msg 209, Level 16 State 1, Line 1
Ambiguous column name 'emp_ID'.
Server: Msg 209, Level 16 State 1, Line 1
Ambiguous column name 'emp_ID'.
```

因为系统不能确定查询的 WHERE 子句是想使用来自 CUSTOMERS 表的 EMP_ID 列还是想要使用来自 EMPLOYEES 表的 EMP_ID 列中的数据, 或是使用两者。

为了使用来自列名出现在不止一个查询的源表中的数据, 必须使用以下形式的合格列名:

<table name>.<column name>

因而, 为了改正上面例子中这种对 EMP_ID 的模糊引用, 可将查询改写为:

```
SELECT RTRIM(cust_f_name)+' '+cust_l_name AS 'Customer',
RTRIM(f_name)+' '+l_name AS 'Salesperson'
FROM customers, employees
WHERE customers.emp_ID = employees.emp_ID
```

当执行改正后的查询时, DBMS 知道对等号左边的表达式提取来自 CUSTOMERS 表中的 EMP_ID 值, 而对等号右边的表达式提取来自 EMPLOYEES 表中的 EMP_ID 值。

技巧 217 使用带 INTERSECT 运算符的 ALL 关键词在查询结果表中包括重复的行

可以使用 INTERSECT 运算符获得出现在两个或多个查询的所有结果表中的行的清单。例如, 如果在名为 AUTO_INS_CUSTOMERS 表中保存汽车保险顾客的清单, 而在名为 HOME_INS_CUSTOMERS 与 UNION 运算兼容的表中保存着家庭保险顾客的清单, 执行以下查询将生成既列出有汽车保险又有家庭保险的顾客的清单:

```
(SELECT * FROM auto_ins_customers)
INTERSECT
(SELECT * FROM home_ins_customers)
```

INTERSECT 运算符与 UNION 运算符类似, 可从其结果表中消除重复的行。由此, 如果想要获得 18~21 岁且在过去一年中有事故记录和违章罚单的汽车保险顾客的清单, 可执行以下查询:

```
(SELECT cust_ID FROM auto_ins_customers
WHERE age BETWEEN 18 AND 21)
INTERSECT
(SELECT cust_ID FROM traffic_violations
```

```
WHERE date_of_infraction >= (GETDATE() - 365))
INTERSECT
```

```
(SELECT cust_ID FROM auto_claims
WHERE date_of_claim >= (GETDATE() - 365))
```

由于 DBMS 消除了查询结果中的重复行，本例查询中的结果表将列出一次且只列出一次特定的 CUST_ID，而不管他在过去一年是收到一次罚单还是 5 次或 3 次罚单。

如果不想让 DBMS 从根据两个或多个查询结果生成的结果表中消除重复的行，可与 INTERSECT 运算符组合使用 ALL 关键词。例如，为了在结果表中对 18~21 岁顾客在过去一年中每次交通违章或事故赔付列出一次 CUST_ID，可以下面的 INTERSECT 执行查询：

```
(SELECT cust_ID FROM auto_ins_customers
WHERE age BETWEEN 18 AND 21)
INTERSECT ALL
(SELECT cust_ID FROM traffic_violations
WHERE date_of_infraction >= (GETDATE() - 365))
INTERSECT ALL
(SELECT cust_ID FROM auto_claims
WHERE date_of_claim >= (GETDATE() - 365))
```

注意：如果具体的 DBMS，如 MS-SQL Server，不支持 INTERSECT 运算符，此时可使用 AND 布尔运算符向查询的 WHERE 子句中添加测试集成员的子查询以代替每个 INTERSECT <query>对。例如，以下 SELECT 语句：

```
SELECT cust_ID FROM auto_ins_customers
WHERE age BETWEEN 18 and 21
AND cust_ID IN (SELECT cust_ID FROM traffic_violations
WHERE date_of_infraction >=
(GETDATE() - 365))
AND cust_ID IN (SELECT cust_ID FROM auto_claims
WHERE date_of_claim >= (GETDATE() - 365))
```

将生成 18~21 岁的在过去一年中既有交通违章又有汽车保险赔付的顾客的清单，这正像是在本技巧中的第二个 INTERSECT 查询一样。

从本技巧的 NTERSECT ALL 查询中生成的结果表需要联合(UNION)两个非 INTERSECT 查询，如：

```
SELECT cust_ID FROM traffic_violations
WHERE date_of_infraction >= (GETDATE() - 365)
AND cust_ID IN (SELECT cust_ID FROM auto_ins_customers
WHERE age BETWEEN 18 and 21)
AND cust_ID IN (SELECT cust_ID FROM auto_claims
WHERE date_of_claim >= (GETDATE() - 365))
UNION ALL
SELECT cust_ID FROM auto_claims
WHERE date_of_claim >= (GETDATE() - 365)
AND cust_ID IN (SELECT cust_ID FROM auto_ins_customers
WHERE age BETWEEN 18 and 21)
AND cust_ID IN (SELECT cust_ID FROM traffic_violations
WHERE date_of_infraction >=
```

```
(GETDATE() - 365)
ORDER BY cust_ID
```

技巧218 在对非 UNION 兼容表的 INTERSECT 查询中使用 CORRESPONDING 关键词

根据定义，如果两个表有同样数目的列而且一个表中每列的数据类型与另一表中对应列的数据类型相同，则这两个表是 UNION 兼容表。如果两个表是 UNION 兼容的，则可对由以下的两个 SELECT 语句返回的行使用 INTERSECT 运算符（读者已在技巧 172 “使用 INTERSECT 运算符选择出现在所有两个或多个源表中的行”中学习了有关知识），生成既来自 TABLE_A 又来自 TABLE_B 的所有行的结果表：

```
SELECT * FROM table_a
INTERSECT
SELECT * FROM table_b
```

另一方面，如果有两个非 UNION 兼容表，仍然可使用 INTERSECT 运算来找出两个表共同的数据集，只要在 INTERSECT 查询中添加 CORRESPONDING 关键词。例如，假设想要得到既向共和党又向民主党捐款的销售商的清单。只要所有表中有相同列名的每列在各表中也有相同的数据类型，则可按以下查询：

```
SELECT * FROM vendors
INTERSECT CORRESPONDING
SELECT * FROM republican_contributors
INTERSECT CORRESPONDING
SELECT * FROM democrat_contributors
```

将产生下面的结果表：

tax_ID	vendor_name	phone_number
88-5481815	'ABC Corporation'	(748)-254-5565
88-5107204	'XYZ Corporation'	(754)-875-5648

在本例中，3 个表中有 3 个列有相同的名称（TAX_ID、VENDOR_NAME、PHONE_NUMBER），而且只有两行在 3 个表中的这 3 个列中有匹配的值（在执行 INTERSECT CORRESPONDING 查询时，DBMS 只在那些在所有表中有相同列名的列中检查匹配的数据值）。

当想要显示只来自某些匹配列中的数据值而仍然要求所有的匹配对应列有相同的值，可在 INTERSECT CORRESPONDING 查询的一个 CORRESPONDING 关键词之后列出想要显示的列。例如，如果想要只显示 VENDOR_NAME，可将本例中的查询改为：

```
SELECT * FROM vendors
INTERSECT CORRESPONDING
SELECT * FROM republican_contributors
INTERSECT CORRESPONDING (vendor_name)
SELECT * FROM democrat_contributors
```

那么 DBMS 对那些在 TAX_ID、VENDOR_NAME 和 PHONE_NUMBER 组合中有相同值的行只显示其 VENDOR_NAME 值。

技巧 219 使用没有 WHERE 子句的多表联合生成笛卡尔积

只要告诉 DBMS 执行多表查询，系统首先要生成列在 SELECT 语句的 FROM 子句中的源表的笛卡尔积。接着系统使用 WHERE 子句中的搜索条件过滤掉不想要的行。因而，如果想要生成两个或多个表的笛卡尔积，只要忽略通常出现在多表查询中的 WHERE 子句即可。

例如，如果想要获得教师及其所教课程的清单，可执行以下查询：

```
SELECT class_ID,
RTRIM(first_name)+' '+last_name AS 'Instructor'
FROM classes, teachers
WHERE classes.instructor = teachers.ID
```

这就把来自 CLASSES 表中的班级信息的每一行与来自 TEACHERS 表的教师名联合起来，生成如下的结果表：

class_ID	Instructor
English 101	Ishud Reedmour
Composition 101	Wanda Wright
Math 101	Mathew Mattick

如果想要获得班级和教师的所有可能的组合（也就是来自 CLASSES 和 TEACHERS 表的所有选列的笛卡尔积），可将查询改写为：

```
SELECT class_ID,
RTRIM(first_name)+' '+last_name AS 'Instructor'
FROM classes, teachers
```

以产生如下的结果表（笛卡尔积）：

class_ID	Instructor
English 101	Ishud Reedmour
Composition 101	Ishud Reedmour
Math 101	Ishud Reedmour
English 101	Wanda Wright
Composition 101	Wanda Wright
Math 101	Wanda Wright
English 101	Mathew Mattick
Composition 101	Mathew Mattick
Math 101	Mathew Mattick

另外，如果有 3 个表——例如多了一个 STUDENTS 表——只要将其添加到查询的 FROM 子句的源表清单中，再在查询的 SELECT 子句中加入想要在结果表中看到的 STUDENTS 列即可。例如，当 DBMS 执行以下查询时：

```
SELECT class_ID,
RTRIM(first_name)+' '+last_name AS 'Instructor',
RTRIM(students.f_name)+' '+students.l_name AS 'Student'
FROM classes, teachers, students
```

将首先生成 CLASSES 和 TEACHERS 表的笛卡尔积，接着系统再生成前面的笛卡尔积表与 STUDENTS 表的笛卡尔积。如果 STUDENTS 表中只有两个学生名 Ima Pupil 和 Uhara Student，则生成如下结果表：

class_ID	Instructor	Student
English 101	Ishud Reedmour	Ima Pupil
Composition 101	Ishud Reedmour	Ima Pupil
Math 101	Ishud Reedmour	Ima Pupil
English 101	Wanda Wright	Ima Pupil
Composition 101	Wanda Wright	Ima Pupil
Math 101	Wanda Wright	Ima Pupil
English 101	Mathew Mattick	Ima Pupil
Composition 101	Mathew Mattick	Ima Pupil
Math 101	Mathew Mattick	Ima Pupil
English 101	Ishud Reedmour	Uhara Student
Composition 101	Ishud Reedmour	Uhara Student
Math 101	Ishud Reedmour	Uhara Student
English 101	Wanda Wright	Uhara Student
Composition 101	Wanda Wright	Uhara Student
Math 101	Wanda Wright	Uhara Student
English 101	Mathew Mattick	Uhara Student
Composition 101	Mathew Mattick	Uhara Student
Math 101	Mathew Mattick	Uhara Student

注意：两个表的笛卡尔积实际上是由来自所有表所有可能的行组合组成的，而每一行又由所有表中的所有列组成。因而，前面的例子并不生成真实的笛卡尔积，因为其显示了行的所有可能的组合，但每一行上没有所有表的所有列的组合。为了获得真正的笛卡尔积，可使用“所有列”运算符（星号*）或是在查询的 SELECT 子句中列出所有（而不是某些）列名。例如，将前面的查询改写为：

```
SELECT * FROM classes, teachers, students
```

可生成 CLASSES、TEACHERS 和 STUDENTS 表的真正的笛卡尔积。

技巧 220 使用别名（关联名）作为表名的简写

如果一个 SQL 语句中的列名在两个或多个语句的源表中有相同的名称，则必须是合格的列名。正如读者在技巧 216 “在有一个或多个相同列名的联合的多个表中多表查询中使用合格的列名”中所学习的，一个合格的列名包括查询要使用的表名和列名。另外，为了引用别人拥有的表，必须使用完全合格的列名，这也就意味着列引用必须包括列名、表名和表拥有者的用户名。因而，如果想要使用另一用户拥有的表生成生日和周年纪念日的清单（例如 KONRAD 拥有的 ANNIVERSARY_BIRTHDAY 表和 HR_ADMIN 拥有的 EMPLOYEES 表），查询中的列引用可能会有些长：

```
SELECT
  CONVERT(CHAR(12,konrad.anniversary_birthday.next_date,107)
  AS 'Date', hr_admin.employees.emp_ID,
  RTRIM(hr_admin.employees.first_name)+
  ' '+hr_admin.employees.last_name AS 'Employee Name',
  konrad.anniversary_birthday.relationship,
  RTRIM(konrad.anniversary_birthday.first_name)+' '+
```

```

konrad.anniversary_birthday.last_name AS
'Family Member', konrad.anniversary_birthday.event,
CONVERT(INTEGER,DATENAME(year,
konrad.anniversary_birthday.next_date)) -
CONVERT(INTEGER,DATENAME(year,
konrad.anniversary_birthday.first_date)) AS 'Years'
FROM hr_admin.employees, konrad.anniversary_birthday
WHERE konrad.anniversary_birthday.next_date
BETWEEN GETDATE() AND (GETDATE() + 30)
AND hr_admin.employees.emp_ID =
konrad.anniversary_birthday.emp_ID
ORDER BY konrad.anniversary_birthday.next_date,
hr_admin.employees.emp_ID

```

由于键入带长的合格列名或几个对多表都是共同的列引用可能是令人烦心的事，SQL 允许使用别名（或关联名）代替语句中使用的任何或全部表名。为了定义可用来代替表名的别名，只要在语句的 FROM 子句中的表名之后键入别名即可。这样一来，在本例中，下面的 FROM 子句：

```
FROM hr_admin.employees, konrad.anniversary_birthday
```

如果在查询中想要使用字母 e 作为 HR_ADMIN.EMPLOYEES 表的别名，而用 ab 来代替 KONRAD.ANNIVERSARY_BIRTHDAY，可使用以下 FROM 子句：

```
FROM hr_admin.employees e, konrad.anniversary_birthday ab
```

接着就可将本例中的查询改写为：

```

SELECT
CONVERT(CHAR(12,ab.next_date,107) AS 'Date', e.emp_ID,
RTRIM(e.first_name)+' '+e.last_name AS 'Employee Name',
ab.relationship,
RTRIM(ab.first_name)+' '+ab.last_name AS 'Family Member',
ab.event, CONVERT(INTEGER,DATENAME(year,ab.next_date)) -
CONVERT(INTEGER,DATENAME(year,ab.first_date)) AS 'Years'
FROM hr_admin.employees e, konrad.anniversary_birthday ab
WHERE ab.next_date BETWEEN GETDATE() AND (GETDATE() + 30)
AND e.emp_ID = ab.emp_ID
ORDER BY ab.next_date, e.emp_ID

```

对关联名（别名）的惟一限制是，不能使用同一别名来引用列在 FROM 子句中的多个表，而且在同一语句中不能既使用别名又使用长（完全的）表名。

技巧 221 理解 NATURAL JOIN

NATURAL JOIN（自然联合）是一类特殊的等价联合，其中带有将一个表中的所有列与另一表中同名的对应列相比较看是否相等的隐含的 WHERE 子句。因此，在 DBMS 通过自然联合的隐含的 WHERE 子句过滤掉源表的积之后，最后的结果表中只有那些在两个表中具有同名的所有列对具有相等值的联合行。

例如，假设有用如下语句创建的 EMPLOYEES 和 SALES 表：

```

CREATE TABLE employees      CREATE TABLE sales
  (emp_ID    INTEGER,          (sales_date DATETIME,

```



```
f_name    VARCHAR(30),    amount_sold MONEY,  
l_name    VARCHAR(30),    emp_ID      INTEGER,  
office_ID INTEGER)
```

为了获得每个办公室的雇员及其销售额的清单，可提交如下的 NATURAL JOIN：

```
SELECT employees.emp_ID, sales.office_ID, f_name, l_name,  
sales_date, amount_sold  
FROM employees NATURAL JOIN sales
```

当 DBMS 执行查询时，将把来自 EMPLOYEES 表的行与来自 SALES 表中那些在 EMP_ID 列（即在两个表中同名的列）中有匹配值的行联合起来。类似地，如果 EMPLOYEES 表也有 OFFICE_ID 列，本例中的 NATURAL JOIN 查询将联合两个表中那些两对同名的列（EMP_ID 和 OFFICE_ID）都有匹配值的行。

事实上，NATURAL JOIN 与检查两个源表中的同名列值的相等性的 WHERE 子句的等价查询是等价的。由此，可将本例中的 NATURAL JOIN 改写如下：

```
SELECT employees.emp_ID, sales.office_ID, f_name, l_name,  
sales_date, amount_sold  
FROM employees, sales  
WHERE employees.emp_ID = sales.emp_ID
```

或者，如果 EMPLOYEES 表也有一个 OFFICE_ID 列，可改写本例中的 EMPLOYEES 表和 SALES 表的 NATURAL JOIN 如下：

```
SELECT employees.emp_ID, sales.office_ID, f_name, l_name,  
sales_date, amount_sold  
FROM employees, sales  
WHERE employees.emp_ID = sales.emp_ID  
AND employees.office_ID = sales.office_ID
```

注意：如果具体的 DBMS 产品，如 MS-SQL Server，不支持 NATURAL JOIN 运算符，只需用带 WHERE 子句（其中使用 AND 运算符来结合在两个表中的同名列对的相等性的搜索条件）等价联合来达到同样的目的。不管形式如何，要记住的重要事情是，NATURAL JOIN 是一种查询，这种查询仅当两个源表中的同名列对具有相等值时才把行联合起来。因而，如果要使用 NATURAL JOIN，应确保在两个表中相关（可联合的）列有相同的名称，而且所有不相关的列对两个表来说都是惟一的。

技巧 222 理解条件联合

条件联合（condition join）是一种多表查询，其中可使用任何一个关系运算符（>、<、>=、<=、<>和=）把一个表中的列与另一表中的对应（相关）列关联起来。简短地说，条件联合与等价联合类似，但有一点不同，即可在条件联合中使用任何一个关系运算符，而在等价联合中只能使用相等运算符（=）。条件联合和带 WHERE 子句的多表查询之间的差别是，在条件联合中，关联表的搜索条件放在 ON 子句中而不是 WHERE 子句中。

例如，下面的条件联合：

```
SELECT * FROM employees JOIN customers  
ON (salesperson_ID = emp_ID)
```

将生成带有来自 CUSTOMERS 表中的每行与 EMPLOYEES 表中的行联合的行的结果表，其中 EMP_ID 列（来自 EMPLOYEES 表）与 SALESPERSON_ID 列（来自 CUSTOMERS 表）

的值相匹配。由此，本例中的条件联合在功能上与下面的多表查询等价：

```
SELECT * FROM employees, customers
WHERE salesperson_ID = emp_ID
```

类似地，可使用如下的条件联合：

```
SELECT DISTINCT e.emp_ID e.f_name, e.l_name, e.total_sales
FROM NV_employees e JOIN AZ_employees
ON (e.total_sales > AZ_employees.total_sales)
```

来生成 Nevada 办公室中那些 TOTAL_SALES（来自 NV_EMPLOYEES 表）列值至少大于一个 Arizona 销售人员的 TOTAL_SALES（来自 AZ_EMPLOYEES 表）列值的销售人员的清单。或者使用以下多表查询来产生相同的结果：

```
SELECT DISTINCT e.emp_ID e.f_name, e.l_name, total_sales
FROM NV_employees e, AZ_employees
WHERE e.total_sales > AZ_employees.total_sales
```

注意：如果具体的 DBMS 产品不支持条件联合，只要将条件联合改写为带 WHERE 子句的多表查询即可。将条件联合的 ON 子句中的搜索条件放到查询的 WHERE 子句中即可。要记住的重要事情是，条件联合与每一种其他多表查询或联合类似，都使用搜索条件（不管是放在 WHERE 子句中还是 ON 子句中）来过滤掉所有联合的无关的行。DBMS 拒绝联合无关的行，因为这些行的列值不能满足 WHERE 子句（或者是 ON 子句）中的搜索条件。

技巧 223 使用 CROSS JOIN 创建笛卡尔积

两个表的 CROSS JOIN（也称为叉积或笛卡尔积）是第 3 个表，其中包括来自两个交叉联合源表中的所有可能的行对。例如，如果有两个表 TABLE_1 和 TABLE_2，每个表有 2 列 3 行，其 CROSS JOIN：

```
SELECT * FROM table_1 CROSS JOIN table_2
```

将把 TABLE_1 中的每一行与 TABLE_2 中的每一行成对以产生结果表，其中有 4 列和 9 行，如图 12.2 所示。

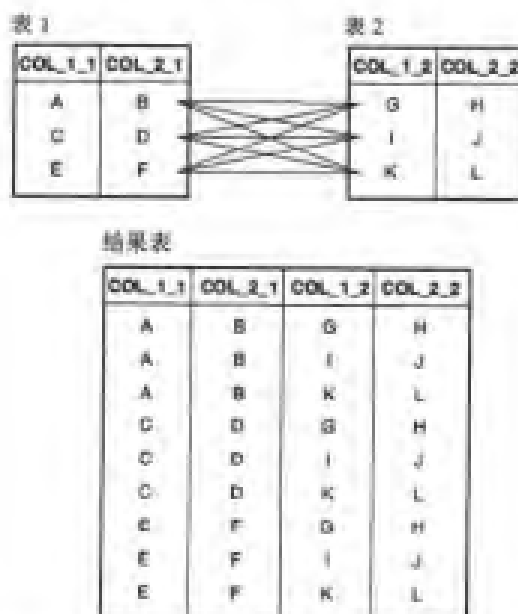


图 12.2 TABLE_1 和 TABLE_2 的 CROSS JOIN

注意：两个表的 CROSS JOIN（或笛卡尔积）中的列数等于第一个表的列数加第二个表的列数。另外 CROSS JOIN 结果表的行数总是等于第一个表的行数乘以第二个表的行数。

如果 DBMS 产品不支持 CROSS JOIN 运算符，只要执行没有 WHERE 子句的多表查询，仍然可以生成两个表的笛卡尔积。例如，假设有一个 ATHLETES 表并想要将每个运算符（athlete）与 DECATHLON_EVENTS（十项运动）表中每个事件行成对。如果 DBMS 支持 CROSS JOIN 运算符，可将查询写为：

```
SELECT * FROM athletes CROSS JOIN decathlon_events
```

或者，使用多表查询生成同样的结果表：

```
SELECT * FROM athletes, decathlon_events
```

正如读者在技巧 210~技巧 216 中的多表查询中所学习的，CROSS JOIN（或笛卡尔积）很少作为查询想要的最后结果。但是，作为每个查询的第一步，DBMS 通常生成 SELECT 语句的 FROM 子句中列出表的 CROSS JOIN。然后系统执行以后的操作和过滤中间（虚拟）笛卡尔积表（源表的）在最后结果表中生成联合的行。

技巧 224 理解列名联合

列名联合（column name join）与读者在技巧 221 “理解 NATURAL JOIN”中所学习的 NATURAL JOIN 非常类似。虽然 NATURAL JOIN 要求在两个表中具有同名的所有列对有匹配的值，但列名联合允许指定 DBMS 要比较的同名列对值。结果，列名联合不仅可用于可用 NATURAL JOIN 联合的表，而且还可用于不能用 NATURAL JOIN 联合的表。毕竟，如果列名联合要求两个表中的同名列对的值相互匹配，那么实际上列名联合就成了 NATURAL JOIN。但是，如果有用如下语句创建的 CUSTOMERS 表和 EMPLOYEES 表：

```
CREATE TABLE customers      CREATE TABLE employees
(cust_ID INTEGER,            (emp_ID INTEGER,
 f_name  VARCHAR(30),        f_name  VARCHAR(30),
 l_name  VARCHAR(30))        l_name  VARCHAR(30))
emp_ID   INTEGER)
```

则不能使用以下的 NATURAL JOIN 来获得显示每个顾客名和顾客的销售人员名的组合清单：

```
SELECT RTRIM(c.f_name)+' '+c.l_name AS 'Customer',
RTRIM(e.f_name)+' '+e.l_name AS 'Salesperson'
FROM customers c NATURAL JOIN employees e
```

正如在技巧 221 中所学习的，NATURAL JOIN 要求所有的同名列对在联合行中有匹配的值。因而，本例中的 CUSTOMERS 和 EMPLOYEES 表的 NATURAL JOIN 将只显示那些与销售人员的姓和姓恰巧分别一样的顾客。

但是，可使用以下的列名联合来联合本例中的 CUSTOMERS 和 EMPLOYEES 表：

```
SELECT RTRIM(c.f_name)+' '+c.l_name AS 'Customer',
RTRIM(e.f_name)+' '+e.l_name AS 'Salesperson'
FROM customers c JOIN employees e
USING (emp_ID)
```

不需要匹配所有同名列对的值，本例查询中的 USING 子句告诉 DBMS 联合那些在 EMP_ID（来自每个表）有匹配值的行——而不管 F_NAME 和 L_NAME 列值是否匹配。

注意：如果具体的 DBMS，如 MS-SQL Server，不支持 USING 子句，可使用条件联合来

代替列名联合。这两种类型的联合的惟一差别是，列名联合隐含地表明在 USING 子句中指定的同名列必须有匹配值，而条件联合在 ON 子句中明确地指明列对的相等。因而，为了将本例中的列名联合转化为条件联合，只需将查询改写为：

```
SELECT RTRIM(c.f_name)+' '+c.l_name AS 'Customer',
       RTRIM(e.f_name)+' '+e.l_name AS 'Salesperson'
FROM customers c JOIN employees e
ON (c.emp_ID = e.emp_ID)
```

技巧 225 使用 INNER JOIN 选择一个表中与另一表中的行相关的所有行

INNER JOIN 是一种多表查询，在这种查询中，DBMS 只返回来自源表中的相关的行对，也就是说，查询的结果表只包括那些满足查询的 ON 子句中的搜索条件的联合的行。作为对照，如果在任一源表中的行在另一表中没有对应（相关）的行，则该行就被过滤掉，不会包括在结果表中。

例如，假设有两个分析家的股票清单，现在想创建两个分析家都推荐的股票的清单。如果一个分析家的清单列在 STOCK_LIST_A 表中，而另一分析家的清单列在 STOCK_LIST_B 中，然后对两个表进行 INNER JOIN，如下所示：

```
SELECT a.symbol,
       a.buy_at AS 'Buy Price A', a.sell_at AS 'Sell Price A',
       b.buy_at AS 'Buy Price B', b.sell_at AS 'Sell Price B'
FROM stock_list_a a INNER JOIN stock_list_b b
ON (a.symbol = b.symbol)
```

这个查询的结果表与下面的类似：

symbol	Buy Price A	Sell Price A	Buy Price B	Sell Price B
CSCO	50	60	55	70
LU	32	40	30	45
F	26	32	27	40
GM	60	69	58	63
VTSS	86	92	82	69
LEN	28	32	30	34

其中只列出既在 STOCK_LIST_A 中又在 STOCK_LIST_B 中的股票简称（SYMBOL）和价格信息。因此，在任一表中带有与另一表中的 SYMBOL 列值不匹配的 SYMBOL 列值的任何行都将被过滤掉，不包括在最后的联合行中。

如果将本例中的 INNER JOIN 改写为等价的多表查询如下：

```
SELECT a.symbol,
       a.buy_at AS 'Buy Price A', a.sell_at AS 'Sell Price A',
       b.buy_at AS 'Buy Price B', b.sell_at AS 'Sell Price B'
FROM stock_list_a a, stock_list_b b
WHERE a.symbol = b.symbol
```

读者将会看到，INNER JOIN 正好是在技巧 210~技巧 216 中学习过的多表等价查询的另一种句法。

注意：在默认情况下，DBMS 将多表查询按 INNER JOIN 来执行，除非指定 OUTER JOIN（此内容将在技巧 227~技巧 229 中学习）之一。

因而，以下查询：

```
SELECT * FROM table_a JOIN table_b  
ON (table_a.column_to_relate = table_b.column_to_relate)
```

与下面的查询等价：

```
SELECT * FROM table_a INNER JOIN table_b  
ON (table_a.column_to_relate = table_b.column_to_relate)
```

技巧 226 理解 USING 子句在 INNER JOIN 中的作用

INNER JOIN（内联合）中的 USING 子句列出那些列值必须匹配的同名列对，以便让 DBMS 在查询的结果表中包括联合的行。换句话说，系统生成 SELECT 语句的 FROM 子句中列出的表的笛卡尔积，然后过滤掉那些列在 USING 子句中没有匹配值的同名列对的联合行。

例如，以下查询中的 USING 子句：

```
SELECT class, section, description, title  
FROM curriculum INNER JOIN book_list  
USING (class)
```

告诉 DBMS 生成只带那些来自 CURRICULUM 表的 CLASS 列值与 BOOK_LIST 表中的 CLASS 列值相匹配的联合行的结果表。

注意：列在 USING 子句中的列名必须出现在 FROM 子句的所有表中。另外，列必须定义为要么有相同的数据类型，要么有相兼容的数据类型。

如果查询中的表间关系是基于多个同名列的匹配值的，则 USING 子句中有不止一个列名。例如，如果班级的每一部分的阅读清单是惟一的，那么在本例中的查询必须改写为：

```
SELECT class, section, description, title  
FROM curriculum INNER JOIN book_list  
USING (class, section)
```

这样才能指明不管 SECTION 列对还是 CLASS 列对必须都具有匹配值，以便让 DBMS 在结果表中包括联合行。

要记住的重要事情是，USING 子句可指明只基于同名列对值相等的表行关系。因而，前面查询中的 USING 子句与以下查询中的 ON 子句等价：

```
SELECT class, section, description, title  
FROM curriculum INNER JOIN book_list  
ON ((curriculum.class = book_list.class) AND  
(curriculum.section = book_list.section))
```

也与以下查询中的 WHERE 子句等价：

```
SELECT class, section, description, title  
FROM curriculum, book_list  
WHERE (curriculum.class = book_list.class)  
AND (curriculum.section = book_list.section)
```

注意：USING 子句实际上并未给 SQL 增加什么功能。由此，许多 DBMS 产品，如 MS-SQL Server，并不支持这一子句。毕竟，正如从本技巧的最后两个示例查询中所看到的，可使用 ON 子句或 WHERE 子句执行与 USING 子句同样的功能。但是，如果具体的 DBMS 支持 USING 子句的话，最好还是使用 USING 子句，因为此子句允许写出比较紧凑的查询，而且（或许）也更易于理解。

技巧 227 理解 OUTER JOIN

不管是内联合还是带 WHERE 子句的多表查询，都组合来自多个表（一次两个表）的行并生成只包括行对的结果表。换句话说，如果任一源表中的行在另一源表中没有匹配（或相关行），DBMS 将不把该行放在最后的结果表中。结果，在执行多表查询或 INNER JOIN 联合时，两个表中不匹配的行看起来就好像消失了一样。

另一方面，OUTER JOIN（外联合）告诉 DBMS 生成结果表，在此表中不仅带有相关行对，而且还有来自两个源表中任一表的不匹配的行。例如，如果有 STUDENTS 表和 FACULTY 表，其中的数据如下：

STUDENTS table			FACULTY table		
f_name	l_name	major	f_name	l_name	dept_head
Sally	Smith	English	Lori	Raines	English
Allen	Winchell	Mathematics	Marcus	Elliot	Engineering
Bruce	Dern	Business	Kelly	Wells	Mathematics
Susan	Smith	NULL	Kris	Matthews	NULL
Howard	Baker	NULL	Linda	Price	NULL

那么下面的单表查询将生成有 5 行数据的结果表：

```
SELECT * FROM students
SELECT * FROM faculty
```

但是，如果将来自 STUDENTS 表中的行与来自 FACULTY 表中的行使用多表查询联合起来，如下所示：

```
SELECT RTRIM(s.f_name+' '+s.l_name AS 'Student', major,
dept_head, RTRIM(f.f_name)+' '+f.l_name AS 'Professor'
FROM students s, faculty f
WHERE major = dept_head
ORDER BY major, dept_head
```

那么 DBMS 将生成只有两行的结果表：

Student	major	dept_head	Professor
Sally Smith	English	English	Lori Raines
Allen Winchell	Mathematics	Mathematics	Kelly Wells

STUDENTS 表中的其余 3 行似乎“消失”了，因为 DBMS 不能通过将 DEPT_HEAD 列值（来自 FACULTY 表）与 MAJOR 列值（来自 STUDENTS 表）匹配从而把 FACULTY 表中的行配对。

类似地，由于同样的原因，系统从 FACULTY 表中过滤掉不匹配的行——在 MAJOR 列（来自 STUDENTS 表）中没有与 DEPT_HEAD 列值（来自 FACULTY 表）相匹配的值。

注意：SQL 标准规定，判式 NULL = NULL 的结果为 FALSE。因此，WHERE 子句过滤掉那些由没有主修的学生（也就是来自 STUDENTS 表的在 MAJOR 列上值为 NULL 的行）与不是系主任的教职员（也就是来自 FACULTY 表的在 DEPT_HEAD 列有 NULL 值的行）联合的所有行。

为了不仅列出来自任一表中的成对（相关）行，还列出不匹配的行，可用 FULL OUTER

JOIN 改写多表查询（事实上，也就是基于在 WHERE 子句中列出的列的 INNER JOIN），如下所示：

```
SELECT RTRIM(s.f_name+' '+s.l_name AS 'Student', major,
dept_head, RTRIM(f.f_name)+' '+f.l_name AS 'Professor'
FROM students s FULL OUTER JOIN faculty f
ON (major = dept_head)
ORDER BY major, dept_head
```

上面的查询可生成如下结果表：

Student	major	dept_head	Professor
NULL	NULL	NULL	Kris Mathews
NULL	NULL	NULL	Linda Price
Susan Smith	NULL	NULL	NULL
Howard Baker	NULL	NULL	NULL
NULL	NULL	Engineering	Marcus Elliot
Bruce Dern	Business	NULL	NULL
Sally Smith	English	English	Lori Raines
Allen Winchell	Mathematics	Mathematics	Kelly Wells

FULL OUTER JOIN 的结果表有 8 行：

- 由多表查询（INNER JOIN）产生的两个联合（相关）行（Sally Smith 和 Allen Winchell）。
- 表示没有主修的学生（Susan Smith 和 Howard Baker）的两个不匹配行（来自 STUDENTS 表）。
- 表示有主修（MAJOR 值为 Business）而在 FACULTY 表中没有系主任的学生（Bruce Dern）的一个不匹配行（来自 STUDENTS 表）。
- 表示不是系主任的教授（Kris Mathews 和 Linda Price）的两个不匹配行（来自 FACULTY 表）。
- 表示是系（Engineering）主任但没有主修学生的教授（Marcus Elliot）的一个不匹配的行（来自 FACULTY 表）。

请注意，DBMS 用每个不匹配行的 NULL 值填入来自“其他”表的其余结果表列。例如，Bruce Dern 是主修商业（business）的。但由于在 FACULTY 表中商业系在 DEPT_HEAD 列无值，于是 DBMS 在结果表的联合行中将 NULL 值放入 DEPT_HEAD 和 PROFESSOR 列（来自 FACULTY 表）。类似地，Kris Mathews 不是系主任，因而 DBMS 在结果表的联合行中的 STUDENT 和 MAJOR 列（来自 STUDENTS 表）放入 NULL 值。

技巧 228 理解 LEFT (RIGHT) OUTER JOIN

正如读者在技巧 227 “理解 OUTER JOIN”中所学习的，INNER JOIN 的结果表只包括相关的行对，而 OUTER JOIN 的结果表既包括匹配的（相关）行也包括不匹配的行。在技巧 227 中 FULL OUTER JOIN 在最后的結果表中包括来自两个表中的不匹配的行。如果想要包括只来自涉及联合（或多表查询）的两个源表中的一个表中的不匹配的行，既可使用 LEFT OUTER JOIN 也可使用 RIGHT OUTER JOIN，而不必使用 FULL OUTER JOIN。

LEFT OUTER JOIN 告诉 DBMS 生成包括联合行和任何不匹配的行的结果表，但不匹配的

行系来自查询的 FROM 子句中 JOIN 关键词左边（LEFT）的表中。作为对照，RIGHT OUTER JOIN 告诉 DBMS 生成包括所有相关行，以及来自 SELECT 语句的 FROM 子句的关键词 JOIN 右边（right，也就是跟在关键词 JOIN 之后）的表中的不匹配行的结果表。

我们来看一个 LEFT OUTER JOIN 的例子，假设想要将来自 CUSTOMERS 表中的顾客姓名和总购货量与在 EMPLOYEES 表中指定的顾客的销售人员联合起来。但是，由于想要的是所有顾客的清单，因而还要包括当前还未指定销售人员的顾客。

如果执行下面的 INNER JOIN：

```
SELECT cust_ID, RTRIM(c.f_name)+' '+c.l_name AS 'Customer',
total_purchases, emp_ID,
RTRIM(e.f_name)+' '+e.l_name AS 'Salesperson'
FROM customers c JOIN employees e
ON (salesperson = emp_ID)
ORDER BY emp_ID DESC
```

结果表将只显示姓名、总购货量和当前指定给顾客的销售人员的姓名（即来自 CUSTOMERS 表中那些可在 EMPLOYEES 表的 EMP_ID 列中找到的 SALESPERSON 值）。结果，任何在 SALESPERSON 列有 NULL 值的 CUSTOMERS 表行都将不出现在 SELECT 语句的结果表中。

但是如果将相同的基本查询按下面的 LEFT OUTER JOIN 来执行的话：

```
SELECT cust_ID, RTRIM(c.f_name)+' '+c.l_name AS 'Customer',
total_purchases, emp_ID,
RTRIM(e.f_name)+' '+e.l_name AS 'Salesperson'
FROM customers c LEFT OUTER JOIN employees e
ON (salesperson = emp_ID)
ORDER BY emp_ID DESC
```

DBMS 将生成下面的结果表：

cust_ID	Customer	total_purchases	emp_ID	Salesperson
6753	Sally Brown	95658.0000	NULL	NULL
3758	Richard Stewart	15425.0000	NULL	NULL
1001	Linda Reed	158112.0000	101	Konrad King
7159	Walter Fields	96835.0000	101	Konrad King
4859	Sue Coulter	45412.0000	101	Konrad King
2158	Jimmy Tyson	754515.0000	201	Kris Jamsa
5159	James Herrera	74856.0000	201	Kris Jamsa

其中不仅包括表明每个顾客及其销售人员的姓名，而且还包括还未指定销售人员的顾客或者其 SALESPERSON 列（来自 CUSTOMERS 表）与 EMP_ID 列（来自 EMPLOYEES 表）值不匹配的顾客。请注意，DBMS 在结果表的每个来自 CUSTOMERS 表中不匹配行中的 EMP_ID 和 SALESPERSON 列（来自 EMPLOYEES 表）中放入 NULL 值。

注意：LEFT OUTER JOIN 中的关键词 LEFT 告诉人们，结果将包括来自查询的 FROM 子句中的关键词 JOIN 左边的表中的不匹配的行。由此，如果将本例中的 FROM 子句改变为：

```
FROM (employees e JOIN customers c)
```

就会得到不同的结果表。新的结果表，就像本例中的结果表一样，仍然有同样的联合行

对。但是，不是不匹配的 CUSTOMERS 表行，结果表将包括所有来自 EMPLOYEES 表中的不匹配的而且在 CUSTOMERS 表中没有匹配的行。

技巧 229 理解 FULL OUTER JOIN

FULL OUTER JOIN 组合了 LEFT OUTER JOIN（读者已在技巧 228“理解 LEFT (RIGHT) OUTER JOIN”中学习过有关知识）的结果表。当 DBMS 执行 FULL OUTER JOIN 时，它生成的结果表中包括联合（相关）的行以及来自 SELECT 语句的 FROM 子句的 JOIN 关键词的右边和左边的表中不匹配的行。

例如，为了列出来自 CUSTOMERS 表的顾客表以及所有来自 FREEZER_INVENTORY 表中的冰箱，可执行下面的 FULL OUTER JOIN：

```
SELECT RTRIM(f_name)+' '+l_name AS 'Customers Name',
customers.freezer_ID AS 'Cust_FID',
freezer_inventory.freezer_ID AS 'Inv_FID',
date_purchased AS 'Purchased', cost,
amt_repairs AS 'Repairs'
FROM customers FULL OUTER JOIN freezer_inventory
ON (customers.freezer_ID = freezer_inventory.freezer_ID)
ORDER BY freezer_inventory.freezer_ID
```

此查询生成的结果表如下：

Customer Name	Cust_FID	Inv_FID	Purchased	Cost	Repairs
Jimmy Tyson	754519	NULL	NULL	NULL	NULL
Linda Reed	158112	NULL	NULL	NULL	NULL
NULL	NULL	11111	2000-10-11	155.9900	10.0000
Richard Stewart	15425	15425	1999-01-01	179.9400	.0000
NULL	NULL	15915	1998-05-05	133.4500	.0000
NULL	NULL	16426	1998-07-05	100.4500	12.7500
NULL	NULL	21345	1996-09-09	100.4500	12.2300
NULL	NULL	22222	2000-04-07	255.5800	.0000
Sue Coulter	45412	45412	1995-03-05	179.9400	45.8900
NULL	NULL	45413	1999-01-01	255.2800	.0000
NULL	NULL	74845	1997-04-01	99.9900	.0000
James Herrera	74856	74856	1999-05-09	185.2500	12.2500
Sally Brown	95658	95658	2000-06-01	188.8500	15.5500
Walter Fields	96835	96835	2000-10-15	155.9900	75.5500
NULL	NULL	97999	1996-09-03	75.9800	44.2500

如果正在使用 FULL OUTER JOIN 不仅要显示两个表中的所有行（既有匹配的也有不匹配的），而且还要查找相关表中的不一致性，一定要在结果表中包括用于将两个表关联起来的列对中的所有列。例如，如果本例中的结果表只显示来自 FREEZER_INVENTORY 表中的 FREEZER_ID，而不能显示 Jimmy Tyson 和 Linda Reed 在 CUSTOMERS 表中的行在 FREEZER_ID 列中有不合法的冰箱 ID 值。没有 CUST_FID 列（来自 CUSTOMERS 表的 FREEZER_ID）的结果表只能表明那两个顾客没有与 FREEZER_INVENTORY 表中的 FREEZER_ID 之一相匹配的 FREEZER_ID 号的冰箱。

作为对照，如果结果表中只有 CUST_FID（来自 CUSTOMERS 表的 FREEZER_ID）而没有 INV_FID 列（来自 FREEZER_INVENTORY 表的 FREEZER_ID 列），仍然可以说明 Jimmy 和 Linda 有不合法的冰箱 ID 号（来自 CUSTOMERS 表的有不合法的 FREEZER_ID 的不匹配行在结果表中的 CUST_FID 列有非 NULL 值，而在 PURCHASED、COSTS 和 REPAIRS 列有 NULL 值）。但是，如果没有 INV_FID 列，结果表只能说明那些当前已借出的冰箱的 FREEZER_ID 值，因为 DBMS 会在每个来自 FREEZER_INVENTORY 表的不匹配行的 CUST_FID 列上放入 NULL 值。

技巧 230 理解 MS-SQL Server 的 OUTER JOIN 记号

读者在技巧 228~技巧 229 中学习如何编写 LEFT OUTER JOIN、RIGHT OUTER JOIN 和 FULL OUTER JOIN 查询时，所有示例都使用了 ON 子句来表明联合匹配行对的关系。但是，MS-SQL Server 还允许将 LEFT 和 RIGHT OUTER JOIN 查询编写为带 WHERE 子句的多表查询。

如果在 SELECT 语句的 WHERE 子句中的比较运算符上接一个星号（*），则 DBMS 将多表查询看作是 OUTER JOIN。正如在表 12.1 中所表明的，星号（*）的位置（是在比较运算符的左边或右边）告诉 DBMS 想要执行的 OUTER JOIN 的类型。

表 12.1 MS-SQL Server 的 WHERE 子句的 OUTER JOIN 记号

OUTER JOIN 的类型	WHERE 子句中的运算符
LEFT OUTER JOIN	*=, *<, *>, *<=, *>=, *<>
RIGHT OUTER JOIN	=*, <*, >*, <=*, >=*, <>*

例如，可把用在技巧 228 “理解 LEFT (Right) OUTER JOIN” 中的 LEFT OUTER JOIN 示例写成多表查询，只要在查询的 WHERE 子句中的比较运算符等号(=)的左边放一星号(*)：

```
SELECT cust_ID, RTRIM(c.f_name)+' '+c.l_name AS 'Customer',
total_purchases, emp_ID,
RTRIM(e.f_name)+' '+e.l_name AS 'Salesperson'
FROM customers c, employees e
WHERE salesperson *= emp_ID
ORDER BY emp_ID DESC
```

类似地，如果在 SELECT 语句的 WHERE 子句中的比较运算符等号(=)的右边放一星号(*)，MS-SQL Server 将执行 RIGHT OUTER JOIN。

技巧 231 在单一查询中联合两个以上的表

不管是执行 OUTER JOIN 还是 INNER JOIN，DBMS 总是一次执行两个表的联合。因而，为了在单一查询中联合 3 个或 3 个以上的表，就需要多个结合源表对、联合（中间）的表对或是单个源表与中间的联合表对的 JOIN 子句。

例如，假设在自己拥有的 PORTFOLIO 表中保存着有关股票的信息，而且想要了解权威人士的交易活动（在 INSIDER_TRADES 表中注明的）是否与分析家的推荐（保存在 ANALYST_RECOMMENDATIONS 表中）完全相关。为了回答这一问题，DBMS 必须联合三个表，一次联合两个。首先写出如下两个表的 INNER JOIN 以获得 PORTFOLIO 表中的那些在 INSIDER_TRADES 表中也有权威人士交易活动的股票：

```
SELECT p.symbol, trade_type, share_ct, position
FROM portfolio p INNER JOIN insider_trades it
ON p.symbol = it.symbol
ORDER BY p.symbol
```

这个 INNER JOIN 告诉 DBMS 从 PORTFOLIO 表中过滤掉那些在 INSIDER_TRADES 表中没有相关行(通过匹配 SYMBOL 列值)的所有行,反之亦然。接着,添加 LEFT OUTER JOIN,将中间表中的联合的 (PORTFOLIO/INSIDER_TRADES) 行与 ANALYST_RECOMMENDATIONS 表中的行相关联(通过 SYMBOL 列值):

```
SELECT p.symbol, trade_type, share_ct, position,
recommendation
FROM portfolio p INNER JOIN insider_trades it
ON p.symbol = it.symbol
LEFT OUTER JOIN analyst_recommendations ar
ON p.symbol = ar.symbol
ORDER BY p.symbol
```

注意: 在本例中的 LEFT OUTER JOIN 告诉 DBMS 让查询结果包括所有来自 ANALYST_RECOMMENDATIONS 表中的匹配行以及来自 PORTFOLIO 表和 INSIDER_TRADES 表的 INNER JOIN 中的不匹配的联合行。如果使用了 INNER JOIN 而不是 LEFT OUTER JOIN, 那么最后的结果表就会只包括来自 PORTFOLIO 表中那些既有权威人士交易(来自 INSIDER_TRADES 表)也有分析家推荐(来自 ANALYST_RECOMMENDATIONS 表)的股票。但是,原始的查询是“分析家的推荐与权威人士的交易相关吗?”因而,真正需要的是不仅包括权威人士的交易与分析家的推荐匹配的,而且还包括所有权威人士的交易而没有分析家推荐的股票的结果表。

在 DBMS 将所有权威人士的交易股票放入最后的结果表中之后,人们可比较权威人士的交易与分析家的推荐相匹配的数目以及不相匹配的数目。

现在,假设想要提交一个查询,只通过联合多个联合的表而不是通过将源表与联合的表(如前例所示)联合起来回答问题。例如,为了获得 PORTFOLIO 表中那些在 INSIDER_TRADES 表中有匹配行或是在 ANALYST_RECOMMENDATIONS 表中有匹配行的股票清单,可提交以下查询:

```
SELECT (CASE WHEN p1.symbol IS NULL THEN p2.symbol
ELSE p1.symbol
END) AS 'Stock',
trade_type, share_ct, position, recommendation
FROM (portfolio p1 INNER JOIN insider_trades it
ON p1.symbol = it.symbol)
FULL OUTER JOIN
(portfolio p2 INNER JOIN analyst_recommendations ar
ON p2.symbol = ar.symbol)
ON p1.symbol = p2.symbol
ORDER BY Stock
```

上面语句先执行 PORTFOLIO 表与 INSIDER_TRADES 表的 INNER JOIN 得出中间结果表 1,再执行 PORTFOLIO 表和 ANALYST_RECOMMENDATIONS 表的 INNER JOIN 得出中间

结果表 2，然后再执行中间结果表 1 和 2 之间的 FULL OUTER JOIN。

要理解的重要事情是，DBMS 总是一次执行两个表的联合（JOIN）。但是要让 DBMS 联合的两个表可以是两个单独的源表、一个单独的源表和联合后的表或者是两个联合后的表。因此，查询可联合任意数目的表，只要了解 DBMS 一次总是联合两个表，从而逐步地将所有表都联合起来。

技巧 232 理解非相等的 INNER 和 OUTER JOIN 语句

技巧 225~技巧 229 中所有的 INNER JOIN 和 OUTER JOIN 示例都使用了相等比较运算符将一个表中的行与另一表中的行根据匹配的列值联合起来。但是，不管 INNER JOIN 还是 OUTER JOIN 查询还都允许根据相等之外的列值来联合行对。

例如，假设一条航线想要生成顾客及其根据每个顾客的经常飞行账户余额而获得的回扣的清单。如下的 INNER JOIN 将生成列出顾客 ID、姓名、账户余额以及那些符合至少可获得一次回扣的顾客所应获得回扣的结果表：

```
SELECT member_ID,  
RTRIM(f_name)+' '+l_name AS 'Member Name', miles_earned,  
miles_required, description AS 'Reward Earned'  
FROM frequent_fliers INNER JOIN rewards  
ON miles_earned >= miles_required  
ORDER BY l_name, f_name, member_ID, miles_required
```

在本例中，FREQUENT_FLIER 表中的行与 REWARDS 表中的行之间的关系并不单纯基于匹配的 MILES_EARNED 和 MILES_REQUIRED 列值。而是最后的结果表将不仅包括那些该两列有匹配值的行，而且还包括那些 MILES_EARNED 列值大于 MILES_REQUIRED 列值的行。在 OUTER JOIN 中还可以使用非相等的比较运算符。例如，如果房地产公司想要准备其资产和有希望的购房者的清单，下面的 FULL OUTER JOIN 查询将生成结果表不仅将每个潜在的购房者与其购房价格范围和房屋规格匹配起来，而且还列出了没有出卖前景的资产以及所列资产不能满足购房者要求的购房者的清单：

```
SELECT address, RTRIM(f_name)+' '+l_name AS 'Buyer',  
min_sales_price AS 'Seller Minimum',  
max_purchase_price AS 'Buyer Maximum',  
(max_purchase_price - min_sales_price) AS 'Spread'  
FROM listings FULL OUTER JOIN buyers  
ON (max_purchase_price >= min_sales_price)  
AND (size_required <= square_footage)  
AND (bedrooms_required <= num_bedrooms)  
ORDER BY address DESC, buyer
```

要理解的重要事情是，在 INNER 和 OUTER 联合的 ON 子句中可使用任何的比较运算符（=、<、>、<=、>=和<>）来表达相关行的列对之间的关系。

技巧 233 理解 UNION JOIN

UNION JOIN 与在技巧 221~技巧 229 中所学习的其他联合不同，此种联合并不试图匹配，实际上是把来自一个源表中的行与来自另一源表中的一行或多行联合起来。这种联合只是创建

包括来自第一个表中列的行加上来自第二个表中的列的行的结果表。

例如以下的 UNION JOIN 将生成带有来自 PORTFOLIO_A 和 PORTFOLIO_B 表的所有行和列的结果表，如图 12.3 所示。

```
SELECT * FROM portfolio_a a UNION JOIN portfolio_b b
```

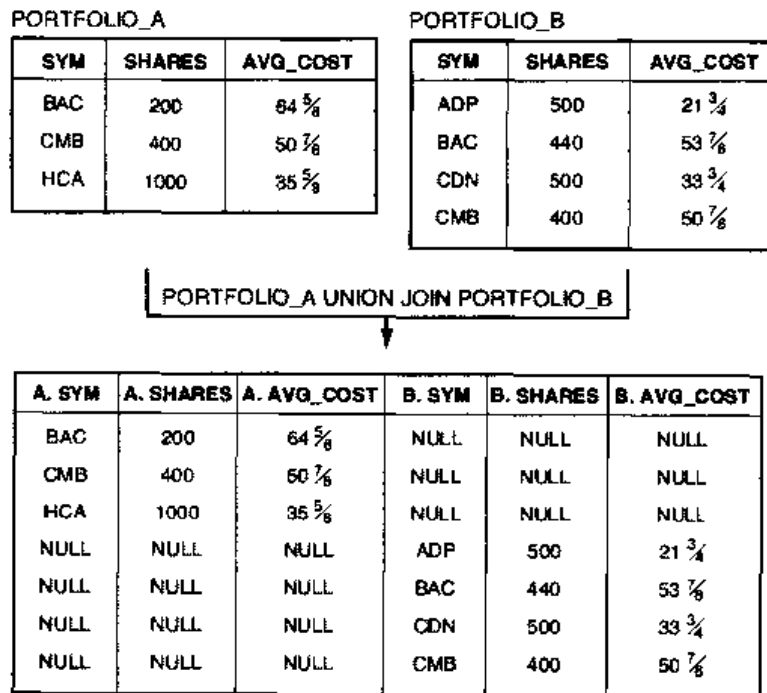


图 12.3 两表的 UNION JOIN 的源表和结果表

虽然 OUTER JOIN 只为不匹配行中的其他表列提供 NULL 值，但 UNION JOIN 中的每一行由来自一个表中的列值与来自另一表中的 NULL 列值联合而组成。在当前的 UNION JOIN 示例中，DBMS 在结果表中插入来自 PORTFOLIO_A 的所有行——并与 PORTFOLIO_B 中每一列都为 NULL 值的行联合成为一行。接着系统将来自 PORTFOLIO_B 的每一行与各列均为 NULL 值的 PORTFOLIO_A 的一行联合起来插入到结果表中。

当想要像单一表一样使用来自两个或多个（或许是与 UNION 不兼容的）表中的所有行时，UNION JOIN 是个便利的工具，这时不会丧失指明哪行来自哪个表的能力。

注意：两个表的 UNION JOIN 的结果表与使用 UNION 运算符选择（SELECT）出现在同样的两个表中的任一或是两个表中的所有行所得到的结果表是不一样的。（读者在技巧 159 “使用 UNION 运算符选择出现在任一或全部的两个或多个表中的所有行”中学习了有关知识。）与显示在图 12.3 中的 UNION JOIN 的结果表不同，同样的两个表的以下 UNION 查询：

```
SELECT * FROM portfolio_a
UNION
SELECT * FROM portfolio_b
```

生成下面的结果表：

```
SYM SHARES AVG_COST
-----
BAC 200    64 5/16
CMB 400    50 7/8
```

```
HCA 1000    33 5/8
ADP 500     21 3/4
BAC 440     53 7/16
CDN 500     33 3/4
```

虽然 UNION JOIN 的结果表有 7 行，每行有 6 列，但当使用 UNION 运算符组合两个表中的行时，生成的是每行 3 列的 6 行。

技巧 234 使用 COALESCE 表达式改善 UNION JOIN 的结果

在技巧 233 “理解 UNION JOIN”中，读者了解到，两个（或多个）表的 UNION JOIN 生成带有来自所有源表的行和列的结果表。结果表中的每行包括来自源表中的一行的列值与来自查询中的另一源表中的每行的每列为 NULL 的联合行。由此 UNION JOIN 生成最后的结果表中有大量的 NULL 值。

例如，以下的 UNION JOIN 查询：

```
SELECT * FROM
joint_acct j UNION JOIN SEP_acct s UNION JOIN IRA_acct i
```

将生成下面的结果表：

sym	shares	avg_cst	sym	shares	avg_cst	sym	shares	avg_cst
BAC	200	64.625	NULL	NULL	NULL	NULL	NULL	NULL
CMB	400	50.875	NULL	NULL	NULL	NULL	NULL	NULL
HCA	1000	35.625	NULL	NULL	NULL	NULL	NULL	NULL
NULL	NULL	NULL	ADP	500	21.75	NULL	NULL	NULL
NULL	NULL	NULL	BAC	440	53.875	NULL	NULL	NULL
NULL	NULL	NULL	CDN	500	33.75	NULL	NULL	NULL
NULL	NULL	NULL	NULL	NULL	NULL	F	500	41.125
NULL	NULL	NULL	NULL	NULL	NULL	HCA	300	27.375
NULL	NULL	NULL	NULL	NULL	NULL	CMB	400	50.875

最后结果表中的大量的 NULL 值使得从其内容中辨别任何有意义的信息变得困难。这时使人想起了一句俗语“只见树木不见森林”。

幸运的是，可使用 COALESCE 表达式（读者在技巧 80 “使用 COALESCE 表达式代替 NULL 值”中学习过有关知识）从结果表中过滤掉 NULL 值。例如，如果将前面的 UNION JOIN 改写为：

```
SELECT
COALESCE (j.symbol, s.symbol, i.symbol) AS 'Symbol',
COALESCE (j.account, s.account, i.account) AS 'Account',
COALESCE (j.shares, s.shares, i.shares) AS 'Shares',
COALESCE (j.avg_cst, s.avg_cst, i.avg_cst) AS 'Avg Cost'
FROM (SELECT 'Joint Acct' AS 'Account', j.*) j,
UNION JOIN (SELECT 'SEP Acct' AS 'Account', s.*) s,
UNION JOIN (SELECT 'IRA Acct' AS 'Account', i.*) i
ORDER BY symbol, account
```

查询的 COALESCE 表达式将从每个列集中选择一个非 NULL 值——因而从 UNION JOIN 子句的中间结果表中过滤掉 NULL 值，以生成类似于下面的结果表：

Symbol	Account	Shares	Avg Cost
ADP	SEP Acct	500	21.7500
BAC	Joint Acct	200	64.6250
BAC	SEP Acct	440	53.8750
CDN	SEP Acct	500	33.7500
CMB	IRA Acct	400	50.8750
CMB	Joint Acct	400	50.8750
F	IRA Acct	500	41.1250
HCA	IRA Acct	300	27.3750
HCA	Joint Acct	1000	35.6250

另外，只需将 ORDER BY 子句改为：

```
ORDER BY account, symbol
```

就可对股票清单根据账户排序——因而更易于看出手中的哪个股票属于哪个账户。

技巧 235 理解 FROM 子句在 JOIN 语句中的作用

JOIN 或是多表 SELECT 语句中的 FROM 子句是虚拟表，DBMS 从此表中提取列值的行，以用作查询中的其余子句的输入。例如，假设有一个 GRADES 表和一个 STUDENTS 表，是由下面的语句创建的：

```
CREATE TABLE grades          CREATE TABLE students
  course_ID    VARCHAR(15),    (SID    INTEGER,
  section      SMALLINT,      f_name VARCHAR(20),
  student_ID   INTEGER,      l_name VARCHAR(20))
  professor_ID INTEGER,
  grade        NUMERIC)
```

如果提交以下的 INNER JOIN 查询：

```
SELECT RTRIM(f_name)+' '+l_name AS 'Student', course_ID,
section, grade
FROM grades JOIN students ON student_ID = SID
```

DBMS 将生成列出学生姓名、班级和成绩的结果表。

当执行一个查询时，DBMS 首先通过联合列在 FROM 子句中那些满足 ON 子句的搜索条件的行生成虚拟表。在本例中，DBMS 生成虚拟表，其中包括来自 GRADES 表中的行与来自 STUDENTS 表中那些 SID 列值与 GRADES 表中的 STUDENT_ID 列值匹配的行联合起来的行。

接着，系统将虚拟表（而不是物理源表的每行）中每一联合行的列值传递给 SELECT 子句，由其过滤掉不想要的列并在结果表中显示其余列。在本例中，SELECT 子句从 GRADES+STUDENTS 虚拟表的联合行中过滤掉除了 F_NAME、L_NAME、COURSE_ID、SECTION 和 GRADE 列值以外的所有列。

类似地，如果在 PROFESSORS 表中有学校教师的姓名，可使用以下查询在班级 ID、学生姓名和成绩旁边显示上课的教授的姓名：

```
SELECT RRIM(s.f_name)+' '+s.l_name AS 'Student', course_ID,
section, grade, RTRIM(p.f_name)+' '+p.l_name AS 'Teacher'
FROM (grades JOIN students s ON student_ID = SID)
JOIN professors p ON professor_ID = PID
```

在本例中, FROM 子句告诉 DBMS 生成一个虚拟表, 其中有来自 STUDENTS 和 GRADES 表的联合的行, 正如前面的示例中所做的那样。接着 DBMS 通过将来自 PROFESSORS 表的行与 STUDENTS+GRADES 虚拟表中的联合后的行联合起来生成另一虚拟表, 该虚拟表中包括那些 PROFESSOR 表中的 PID 列值与 STUDENTS+GRADES 虚拟表中的联合行的 PROFESSOR_ID 列值相匹配的行。

因而, 虽然在下面查询中的 FROM 子句:

```
SELECT * FROM students, grades
WHERE student_ID = SID
ORDER BY student_ID
```

给人的印象好像是, FROM 子句不过是列出 SELECT 语句使用的源表名而已, 但实际上, FROM 子句是包括作为查询数据源的联合行的虚拟表。DBMS 首先既可生成来自列在 FROM 子句中的表的笛卡尔积的虚拟表, 也可通过联合匹配 FROM 子句中的每个 JOIN 语句中指明的相关表而生成虚拟表。接着, 系统使用 SELECT 语句中的其他子句来过滤掉虚拟表中不想要的行和列, 并把余下的列值作为行添加到最后的结果表中。

技巧 236 在多表 JOIN 中使用 “*” 运算符指明所有或只是某些表中的所有列

星号 (*) (即“所有列”运算符) 给出了告诉 DBMS 想要在结果表中包括所有表列的一种简捷方式。例如, 下面的 SELECT 语句的结果表将包括 STUDENTS 表中的所有列:

```
SELECT * FROM students
```

类似地, 下面的查询的结果表包括来自 STUDENTS 中的所有列, 后面跟着来自 GRADES 表中的所有列:

```
SELECT * FROM students, grades
WHERE SID = student_ID
```

如果执行 3 个 (或更多) 表的查询, 而在 SELECT 子句中只有所有列运算符星号 (*), 结果表中将包括来自 FROM 子句中的第一个表的所有列, 后面跟着来自 FROM 子句中的第二个表的所有列, 再跟着来自 FROM 子句中的第 3 个表的所有列, 依此类推。

为了显示只来自 FROM 子句中的某些表的所有列, 可使用“合格的”所有列运算符。正如读者在技巧 216 “在有一个或多个相同列名的联合的多个表中多表查询中使用合格的列名”中所学习的, 一个“合格的”列名是包括列所在表名的列名。类似地, “合格的”所有列运算符包括 DBMS 要显示所有列的表名。这样一来, 为了显示来自 STUDENTS 表中的所有列, 可执行如下所示的 SELECT 语句:

```
SELECT students.* FROM students
```

其中使用了合格的所有列运算符 STUDENTS.*, 告诉 DBMS 显示 STUDENTS 表中的所有列。类似地, 如果想要显示 STUDENTS 表中的所有列, 而只显示 GRADES 表中的两个, 可执行如下查询:

```
SELECT students.*, grade FROM students, grades
WHERE SID = student_ID
```

上面的查询使用了合格的所有列运算符, 告诉 DBMS 显示 STUDENTS 表中的所有列并只显示 GRADES 表中的 COURSE_ID 和 GRADE 列。

技巧 237 在单表 JOIN（即自我 JOIN）中使用表别名

听起来有点奇怪，某些多表查询（或联合）涉及表与本身的关系（而不是与另一表的关系）。例如，假设为顾客提供一种激励机制以便将自己的公司推荐给他所认识的人。如果在 CUSTOMERS 表的一列（如 REFERRER 列）上跟踪推荐顾客的 ID 号，可提交以下查询来生成显示推荐人的 CUST_ID 以及由另一顾客推荐的每个顾客的姓名：

```
SELECT cust_ID, RTRIM(f_name)+' '+l_name AS 'Customer',  
referrer AS 'Referred By'  
FROM customers  
WHERE referrer IS NOT NULL  
ORDER BY "Referred By"
```

但是，为了列出每个推荐人的姓名而不是 CUST_ID 号，必须将 CUSTOMERS 表中那些在 REFERRER 列有非 NULL 值的每一行与 CUSTOMERS 表中在 CUST_ID 列中有匹配值的行联合起来。换句话说，必须把 CUSTOMERS 表中的一些行与 CUSTOMERS 表中的另外的行根据 REFERRER 和 CUST_ID 列值的匹配 JOIN 起来。

根据所学过的有关多表查询和 JOIN 语句中 FROM 子句的作用的内容（在技巧 210 “使用 FROM 子句执行多表查询”和技巧 235 “理解 FROM 子句在 JOIN 语句中的作用”中），人们可能会认为通过在 SELECT 语句的 FROM 子句中包括同样的表名两次，即可将一个表内的行与同一表中的其他行联合起来，如下面的语句那样：

```
SELECT cust_ID, RTRIM(f_name)+' '+l_name AS 'Customer',  
RTRIM(f_name)+' '+l_name AS 'Referred By'  
FROM customers, customers  
WHERE referrer = cust_ID  
ORDER BY "Referred By"
```

当执行上面的查询时，DBMS 应该首先将该表与其本身中的行执行 CROSS JOIN。然后系统应该使用 WHERE 子句中的搜索条件从（虚拟的）中间表中过滤掉不想要的联合行，即那些 CUST_ID 列值与 REFERRER 列值不匹配的行。

遗憾的是，SQL 不允许在单一的 FROM 子句中列出同一表名一次以上。结果，当提交上面的示例查询时，系统将显示如下的错误消息：

```
Server: Msg 1013, Level 15, State 1, Line 4  
Tables 'customers' and 'customers' have the same exposed  
names. Use correlation names to distinguish them.
```

另外，如果在 FROM 子句取消第二个 CUSTOMERS 引用，DBMS 将执行该查询。但是，由于系统一次一行地处理 CUSTOMERS 表，所以要从 CUSTOMERS 表行中显示那些 REFERRER 列值与 CUST_ID 列值相等的行中姓名。由于顾客不能推荐自己，查询（在 FROM 子句中只引用 CUSTOMERS 表一次）将成功执行，但并不能提供所希望获得的顾客姓名。

为了将表与自身联合起来，SQL 要求在第二次引用同一表时使用别名或关联名。用这种办法，DBMS 在处理查询的 FROM 子句时就将两个有“不同”表名的表联合起来。这样一来，如下的 SELECT 语句：

```
SELECT customers.cust_ID, RTRIM(customers.f_name)+  
' '+customers.l_name AS 'Customer',  
RTRIM(referrers.f_name)+' '+referrers.l_name
```

```
AS 'Referred By'
FROM customers, customers referrers
WHERE customers.referrer = referrers.CUST_ID
ORDER BY "Referred By"
```

将 REFERRERS 定义为 CUSTOMERS 表的别名，将显示推荐人姓名以及公司现有的该推荐人所推荐的每个顾客的姓名。

在本例中，DBMS 生成 CUSTOMERS 和（假想的、重复的）REFERRERS 表的笛卡尔积。然后系统从中间（虚拟）CUSTOMERS+REFERRERS 表中过滤掉那些 REFERRER 列值（来自 CUSTOMERS 表）不等于 CUST_ID 列值（来自假想的 REFERRERS 表）的行。

类似地，可使用如下查询来获得从推荐人处接收到的顾客数目（在本例中 WHERE 子句从结果表中消除了那些未给出任何推荐的顾客）：

```
SELECT c.cust_ID, RTRIM(c.f_name)+' '+c.l_name
AS 'Customer',
(SELECT count(*) FROM customers r
WHERE r.referrer = c.cust_ID) AS 'Referral Count'
FROM customers c
WHERE c.cust_ID IN (SELECT referrer FROM customers)
ORDER BY "Referral Count" DESC
```

技巧 238 理解表的别名

在技巧 237 “在单表 JOIN（即自我 JOIN）中使用表别名”中，读者已经了解到，只要想执行自身联合（即将本表中的一行与同一表中的其他行联合），SQL 就要求使用第二个表名或别名。但是可在任何查询中使用表的别名（又称为关联名），而不管 SQL 要求不要求。

例如，假设想要从另一用户（SUSAN）拥有的 PROSPECTS 表的行中列出姓名和电话号码。可执行如下查询：

```
SELECT RTRIM(susan.prospects.f_name)+
' '+susan.prospects.l_name AS 'Prospect',
susan.prospects.phone_number
FROM susan.prospects
ORDER BY 'Prospect'
```

始终在此查询中键入了完全合格的列名引用（当引用另一用户拥有的表中的列时，必须指明完全合格的列名[包括表的拥有者的用户名、表名和列名]）。

作为替代方法，可以通过在 SELECT 语句的 FROM 子句中定义表的别名来缩短列名引用：

```
SELECT RTRIM(sp.f_name)+' '+sp.l_name AS 'Prospect',
sp.phone_number
FROM susan.prospects sp
ORDER BY 'Prospect'
```

为了定义表的别名（或关联名），可在 SELECT 语句的 FROM 子句中的表名之后紧跟着键入想要使用的别名。然后在整个查询中都可用别名来代替表名（或完全合格的表名，如本例中的 SUSAN.PROSPECTS）。

使用表别名的好处是，当编写显示两个或多个表的联合中的几个列时（特别是表有较长的名称或为另一用户所拥有时）更为明显。例如，假设读者是个销售经理而且想要生成属于自

己所管理的 3 个销售人员 (FRANK、SUSAN 和 RODGER) 的 PROSPECTS 表中的重复的电话号码。如果执行以下查询:

```
SELECT RTRIM(COALESCE(frank.prospects.f_name,
rodger.prospects.f_name,susan.prospects.f_name))+ ' '+
COALESCE(frank.prospects.l_name,rodger.prospects.l_name,
susan.prospects.l_name) AS 'Prospect',
COALESCE(frank.prospects.phone_number,
rodger.prospects.phone_number,
susan.prospects.phone_number) AS 'Phone Number',
(CASE WHEN frank.prospects.phone_number IS NOT NULL
THEN 'Frank ' ELSE '' END)+
(CASE WHEN rodger.prospects.phone_number IS NOT NULL
THEN 'Rodger ' ELSE '' END)+
(CASE WHEN susan.prospects.phone_number IS NOT NULL
THEN 'Susan' ELSE '' END) AS 'Being Called By'
FROM ((frank.prospects FULL JOIN rodger.prospects ON
frank.prospects.phone_number =
rodger.prospects.phone_number)
FULL JOIN susan.prospects ON
frank.prospects.phone_number =
susan.prospects.phone_number
OR rodger.prospects.phone_number =
susan.prospects.phone_number)
WHERE frank.prospects.phone_number =
rodger.prospects.phone_number
OR frank.prospects.phone_number =
susan.prospects.phone_number
OR rodger.prospects.phone_number =
susan.prospects.phone_number
ORDER BY "Being Called By", 'Prospect'
```

DBMS 将显示下面的结果表:

Prospect	Phone Number	Called By
Bill Barteroma	(222)-222-2222	Frank Rodger Susan
Steve Kernin	(555)-555-5555	Frank Rodger Susan
Frank Burns	(111)-222-1111	Frank Susan
Hawkeye Morgan	(333)-222-3333	Frank Susan
Steve Pierce	(444)-222-4444	Frank Susan
Walter Phorbes	(666)-666-6666	Rodger Susan

对这 3 个表名可使用别名, 可将查询缩短为:

```
SELECT RTRIM(COALESCE(fp.f_name,rp.f_name,sp.f_name))+ ' '+
COALESCE(fp.l_name,rp.l_name,sp.l_name) AS 'Prospect',
COALESCE(fp.phone_number,rp.phone_number,sp.phone_number)
AS 'Phone Number',
(CASE WHEN fp.phone_number IS NOT NULL
THEN 'Frank ' ELSE '' END)+
```

```

    (CASE WHEN rp.phone_number IS NOT NULL
    THEN 'Rodger ' ELSE '' END)+
    (CASE WHEN sp.phone_number IS NOT NULL
    THEN 'Susan' ELSE '' END) AS 'Being Called By'
FROM ((frank.prospects fp FULL JOIN rodger.prospects rp
ON fp.phone_number = rp.phone_number)
FULL JOIN susan.prospects sp
ON fp.phone_number = sp.phone_number
OR rp.phone_number = sp.phone_number)
WHERE fp.phone_number = rp.phone_number
OR fp.phone_number = sp.phone_number
OR rp.phone_number = sp.phone_number
ORDER BY "Being Called By", 'Prospect'

```

正如读者可以看到的，在查询中使用 FP、RP 和 SP（作为别名）代替完全合格的表名，不仅可以使列名引用较短，还减少了令人感到心烦的键入，而且还使查询易于阅读。

技巧 239 理解 ANY 的模糊本质以及 SQL 如何使其表示 SOME

在英语中，单词 any 可有两种不同的意义，到底是哪种意义要依赖于使用的上下文而定。

例如，如果问：“Do any of you know how to write an SQL query?”（你们中的任何人[any]知道如何编写 SQL 查询吗？），此句中的 any 意为一种存在量词，问话想要了解的是是否至少有一个人（在所提到的人中）了解如何编写 SQL 查询。作为对照，如果说：“I have more fun writing SQL statements than any of you.”（我编写 SQL 语句比你们任何人都会获得更多的乐趣）。此句中的 any 是一种广义量词，其意义为“我编写 SQL 语句比读者中的每一个人都有趣。”

SQL 允许与 6 种比较运算符（=、<>、<、<=、>和>=）一起使用关键词 ANY 的存在性的内涵，以便将单个值与由子查询所生成的数据列加以比较。如果至少有一个单个值与子查询生成的数据集中的某一个数据的比较之一求值为 TRUE，那么整个 ANY 测试求值为 TRUE。

例如，如果在 PORTFOLIO 表有股票代码和价格的清单，而在 PRICE_HISTORY 表中有股票简称和历史价格信息的清单，可执行以下查询来获得股票简称以及那些当前价格比同一股票在过去 6 个月（180 天）中的至少一天的收盘价格多 50%或更多的当前价格：

```

SELECT symbol, current_price FROM portfolio
WHERE current_price >= ANY
(SELECT closing_price * 1.5 FROM price_history
WHERE price_history.symbol = portfolio.symbol
AND price_history.trade_date >= (GETDATE() - 180))

```

当执行这一查询时，DBMS 一次一行地测试 PORTFOLIO 表的 CURRENT_PRICE 列的数据值。系统从提取 PORTFOLIO 表中的 SYMBOL 列及其 CURRENT_PRICE 列中的价格。然后子查询生成其中有股票在过去 180 天中的每一天的 CLOSING_PRICE（收盘价）值乘以 1.5（150%）的结果的清单。如果有任何（any，也就是说至少一个）CURRENT_PRICE 值大于或等于 CLOSING_PRICE 清单中的计算值（由子查询生成），则 DBMS 在结果表中包括该股票的 SYMBOL 及其 CURRENT_PRICE 值。如果 CURRENT_PRICE 小于所有的由子查询生成的清单中计算的 CLOSING PRICE * 150%值，那么系统就不在结果表中包括该股票的 SYMBOL 及其 CURRENT_PRICE 值。

同一查询可改写如下:

```
SELECT symbol, current_price FROM portfolio
WHERE current_price >= SOME
(SELECT closing_price * 1.5 FROM price_history
WHERE price_history.symbol = portfolio.symbol AND
price_history.date >= (GETDATE() - 180))
```

这个查询可读作:“请列出那些在过去 6 个月中同一股票的 CURRENT_PRICE 大于或等于 CLOSING_PRICE 值的清单中的某些值的股票简称 (SYMBOL) 和当前价格 (CURRENT_PRICE)。”

注意:虽然 SQL-92 标准指明,关键词 SOME 是关键词 ANY 的另一种用法,但某些 DBMS 产品还不支持使用 SOME。请检查一下系统文档,看具体的 DBMS 是否允许这样做。如果允许,则可使用关键词 SOME 来代替关键词 ANY,因为 SOME 意义较为明确——与 ANY 不同,从没有“全部”的词义。

技巧 240 使用 EXISTS 而不使用 COUNT(*)检查子查询是否至少返回一行

顾名思义, COUNT(*)函数返回一个表中满足 WHERE 子句中的搜索条件的行的计数。而 EXISTS 判式是布尔表达式,如果子查询至少生成有一行的结果表,则求值为 TRUE。因此以下判式:

```
WHERE 0 < (SELECT COUNT(*) FROM <table name>
WHERE <search condition(s)>)
```

与下面的判式是等价的:

```
WHERE EXISTS (SELECT * FROM <table name>
WHERE <search condition>)
```

当第一个 WHERE 子句中的子查询没有返回行时,其 COUNT(*)函数返回 0,而 WHERE 子句求值为 FALSE。类似地,如果用在第二个 WHERE 子句中的同样的子查询没有返回行,根据定义 EXISTS 求值为 FALSE,就使得 WHERE 子句求值为 FALSE。

作为对照,如果在第一个 WHERE 子句中的子查询返回一行或多行,其 COUNT(*)函数将返回大于 0 的值,接着就使得 WHERE 子句求值为 TRUE。类似地,如果在第二个 WHERE 子句中的同样的子查询返回一行或多行,根据定义, EXISTS 求值为 TRUE,这就接着使得 WHERE 子句求值为 TRUE。

因而,如果想要得到在过去的一年中至少有一次事故的投保人的清单,可在如下的查询中使用 COUNT(*)函数:

```
SELECT cust_ID, f_name, l_name FROM customers
WHERE 0 < (SELECT COUNT(*) FROM claims
WHERE claims.date_of_claim >= (GETDATE() - 365)
AND claims.cust_ID = customers.cust_ID)
```

或者,使用有 EXISTS 判式而不使用 COUNT(*) > 0 比较测试下面的类似查询:

```
SELECT cust_ID, f_name, l_name FROM customers
WHERE
EXISTS (SELECT * FROM claims
WHERE claims.date_of_claim >= (GETDATE() - 365)
AND claims.cust_ID = customers.cust_ID)
```

当把这两个查询之一提交给 DBMS 之后，系统一次一行地处理 CUSTOMERS 表并对 CUSTOMERS 表中的每行执行 WHERE 子句中的子查询。

当执行第一个 SELECT 语句中的子查询时，系统对 CLAIMS 表中的那些既有 DATE_OF_CLAIM 同时行中的 CUST_ID 值又与 CUSTOMERS 表中的 CUST_ID 列值相匹配的行计数。如果 COUNT(*)函数返回的值大于 0，则 WHERE 子句求值为 TRUE，于是 DBMS 将该顾客的 CUST_ID 和姓名添加到结果表中。

当处理第二个 SELECT 语句时，只要 DBMS 在 CLAIMS 表发现了满足子查询的 WHERE 子句中的搜索条件的一行就可中止子查询的执行。如果子查询返回任何行，则 EXISTS 判式求值为 TRUE，系统将 CUSTOMERS 表中的该顾客的 CUST_ID 和姓名添加到查询的结果表中。

那么，到底应该使用 EXISTS 还是 COUNT 来决定表中是否至少有一行满足子查询中的 WHERE 子句中的搜索条件呢？简单的回答是，使用 EXISTS。

另外还可看出，使用 EXISTS 来检查是否存在至少一行满足 WHERE 子句中的搜索条件还使得查询比使用 $0 < \text{COUNT}(*)$ 的比较更易于阅读。另外，根据具体的 DBMS 所使用的优化器的能力，EXISTS 可能实际上比 COUNT(*)执行得更快，特别是如果子查询中的表有大量的行时。

请记住，COUNT(*)函数的目的是对表中那些满足查询（或子查询）的 WHERE 子句中的搜索条件的行计数。因此，当使用 COUNT(*)检查是否至少有一行满足查询（或子查询）的 WHERE 子句中的搜索条件时，某些 DBMS 产品将读取查询的表中的每一行，对那些满足 WHERE 子句中的搜索条件的行计数。仅当完成其全表扫描之后（这是个花时间的操作），COUNT(*)才能返回满足 WHERE 子句中的搜索条件的行数（如果 COUNT(*)返回 0 值，那么 WHERE 子句返回 FALSE，反之亦然）。

另一方面，EXISTS 判式并不使用由子查询的 SELECT *子句返回的任何数据值。另外，DBMS 对子查询表中满足 WHERE 子句中的搜索条件的行数的多少并不介意。因此，只要 DBMS 发现了第一个列值满足子查询中的 WHERE 子句中的搜索条件的行即可停止提取和过滤运算。

简短地说，当要确定是否至少有一行满足子查询中的 WHERE 子句中的搜索条件时，EXISTS 的性能超过了 COUNT(*)函数。如果在有 10,000,000 行的表中的第二行就满足子查询中的 WHERE 子句中的搜索条件，EXISTS 只需要处理两行。而 COUNT(*)函数却必须读取所有 10,000,000 行，然后将满足搜索条件的行的计数与 0 加以比较。

技巧 241 理解何时使用 ON 子句以及何时使用 WHERE 子句

ON 子句总是在 JOIN 子句中用作过滤器，有关 JOIN 子句的内容已在技巧 225~技巧 229 中学习过。同时 WHERE 子句用于过滤掉由单表和多表查询返回的不想要的行。因而，ON 子句（如果有的话）必须总是跟随 JOIN 子句。另一方面，WHERE 子句既可用在带 JOIN 子句的查询中，也可用在没有 JOIN 子句的查询中。

在 INNER JOIN 查询中的 ON 子句在功能上与带同样搜索条件的 WHERE 子句等价。因而，如果想要生成列出每位教师姓名（来自 TEACHERS 表）以及每位教授所教课程的描述（来自 CLASSES 表），下面这个 INNER JOIN（带 ON 子句）：

```
SELECT course_ID, description,  
       RTRIM(f_name) + ' ' + l_name AS Instructor
```

```
FROM classes JOIN teachers
ON classes.professor_ID = teachers.PID
```

与下面的多表 SELECT 语句（带 WHERE 子句）产生同样的结果表：

```
SELECT course_ID, description,
RTRIM(f_name)+' '+l_name AS Instructor
FROM classes, teachers
WHERE classes.professor_ID = teachers.PID
```

作为对照，当执行一种 OUTER JOIN 查询（LEFT、RIGHT 或 FULL OUTER JOIN）时，使用带同样搜索条件的 WHERE 子句而不使用 ON 子句将产生同样的结果表。在 OUTER JOIN 中，不管是 ON 子句还是 WHERE 子句都过滤掉那些不满足 WHERE 子句中的搜索条件的行。但是，OUTER JOIN 中的 WHERE 子句过滤掉的行（也就是排除的）是不包括在结果表中的。同时，OUTER JOIN 中的 ON 子句首先过滤掉不想要的行，然后在结果表中以 NULL 值列包括某些或所有排除的行。

例如，假设 TEACHERS 表中的某些教授还没有指定所教的课程。图 12.4 中显示在 MS-SQL Server Query Analyzer 窗口的上窗格中的 LEFT OUTER JOIN（没有 WHERE 子句）将产生图 12.4 的下窗格中的结果表。

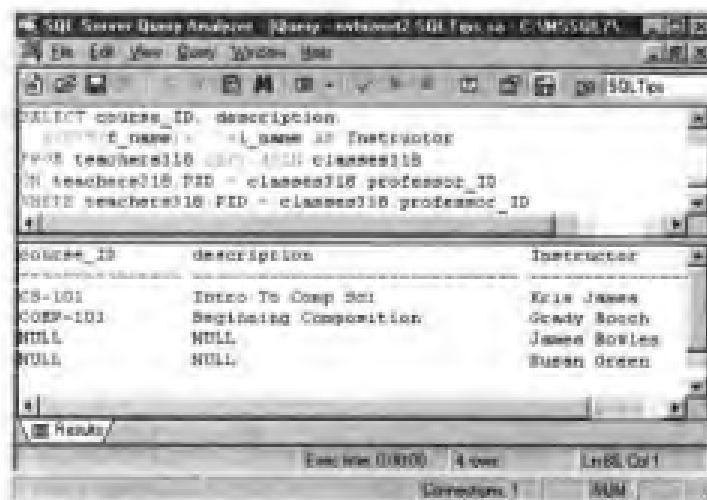


图 12.4 两表的 LEFT OUTER JOIN 的 MS-SQL Server Query Analyzer 窗口

作为对照，下面这个 LEFT OUTER JOIN（带 WHERE 子句）：

```
SELECT course_ID, description,
RTRIM(f_name)+' '+l_name AS Instructor
FROM teachers LEFT JOIN classes
ON teachers.PID = classes.professor_ID
WHERE teachers.PID = classes.professor_ID
```

将产生如下结果表：

course_ID	description	Instructor
CS-101	Introduction To Computer Science	Kris Jamsa
COMP101	Beginning Composition	Grady Booch

这个结果表在任何行上都没有 NULL 值。

技巧 242 理解如何使用嵌套的查询同时处理多个表

子查询或嵌套的查询是在另一 SQL 语句内的 SELECT 语句。虽然嵌套的查询总是 SELECT 语句，但包括子查询的 SQL 语句（其中包括嵌套的 SELECT 语句[或子查询]）可能是 DELETE、INSERT、UPDATE 或另一条 SELECT 语句。事实上，嵌套的查询甚至可以是另一子查询内的子查询。

由于包括有子查询的 SQL 语句及其子查询都可以使用不同的表，有时可使用包括（单表）子查询的（单表）查询来代替（两表）JOIN。例如，当想在查询结果中只包括来自一个表的数据值时，可使用嵌套的查询而不是用 JOIN 把两个表中的信息关联起来。作为一个示例，下面的嵌套的查询将列出在过去的一年中至少有一次股票交易的所有顾客的清单：

```
SELECT CID AS 'Cust ID',  
RTRIM(f_name)+' '+l_name AS 'Customer'  
FROM customers  
WHERE EXISTS (SELECT * FROM trades  
WHERE trade_date >= GETDATE() - 365  
AND cust_ID = CID
```

作为对照，当需要生成带有不止一个表中的数据值的结果表时，常常必须要么执行 JOIN 要么执行带 WHERE 子句的多表查询。

例如，可使用下面的查询，将来自 CUSTOMERS 表中的行与来自 TRADES 表中的行联合起来，生成在过去一年里的那些交易额为 \$100,000 或更多的顾客姓名及其交易额的清单：

```
SELECT trade_date, symbol, shares * price AS 'Total Trade',  
CID, RTRIM(f_name)+' '+l_name AS 'Customer'  
FROM customers JOIN trades  
ON CID = cust_ID  
WHERE shares * price > 100000  
AND trade_date >= GETDATE() - 365  
ORDER BY Customer
```

但是，多表查询和嵌套的子查询并不互相排斥。例如，假设想要包括诸如美元额总计的 SUM() 和交易数的 COUNT() 以及前例中的其他信息。以下查询将执行与前面示例中的相同的两表 JOIN：

```
SELECT trade_date, symbol, shares * price AS 'Total Trade',  
(SELECT COUNT(*) FROM trades  
WHERE trade_date > GETDATE() - 365  
AND cust_ID = CID) AS 'Count',  
(SELECT SUM(price) * SUM(shares),  
FROM trades  
WHERE trades.trade_date >= GETDATE() - 365  
AND cust_ID = CID) AS 'Total $ Volume',  
CID AS 'Cust ID', TRIM(f_name)+' '+l_name AS 'Customer'  
FROM customers JOIN trades  
ON CID = cust_ID  
WHERE shares * price >= 100000  
AND trade_date >= GETDATE() - 365  
ORDER BY Customer
```


但是，查询的结果表包括由两个总计函数子查询（在 SELECT 子句中）返回的值。

简短地说，嵌套查询可与包括子查询的 SQL 语句无关，而且可使用来自列在包括语句的 FROM 子句中的表的任何一列的值。可使用嵌套查询来执行多表操作而不必联合多个相关表中的行。但是，如果需要来自多个表中的数据，或者如果想要在结果表的同一行中有独立的列值以及总计函数值，就可嵌套带多表查询或 JOIN 的 SELECT 子句中所需的总计函数的子查询。

第 13 章 理解 SQL 子查询

技巧 243 理解在子查询中引用时的主查询列的值

对子查询操作的关键特点是，引用主（或包括子查询的）查询中当前行的列值的能力。下面的查询将列出销售部门中每个雇员的 EMP_ID、姓名和总销售额：

```
SELECT emp_ID, RTRIM(f_name) + ' ' + l_name AS 'Name',  
       (SELECT SUM(invoice_total) FROM invoices  
        WHERE salesperson = emp_ID) AS 'Sales Volume'  
FROM employees WHERE dept = 'Sales'
```

因此，查询在结果表的每行上既给出总计也给出详细信息。主（包括）查询提供来自 EMPLOYEES 表的雇员明细（EMP_ID 和姓名），而子查询（或内查询）提供来自 INVOICES 表的销售总量（总计）。

为了生成查询的结果表，DBMS 通过一次读取 EMPLOYEES 表中的一行雇员数据来执行主（包括）查询：

```
SELECT emp_ID, RTRIM(f_name) + ' ' + l_name AS 'Name'  
FROM employees WHERE dept = 'Sales'
```

在提取了雇员信息之后，系统检查当前行（也就是刚刚读入的并正在处理的行）的 DEPT 列，看销售部门是否有该雇员。如果该雇员确是销售部门的，DBMS 就将当前行的 EMP_ID 和姓名添加到结果表中，然后再执行子查询从 INVOICES 表中获取该雇员的总销售额：

```
(SELECT SUM(invoice_total) FROM invoices  
 WHERE salesperson = emp_ID)
```

请注意，在本例中，子查询的 WHERE 子句的 SALESPERSON 引用了子查询的 INVOICES 表中的列，而子查询的 WHERE 子句引用了主查询的 EMPLOYEES 表中的列。在 DBMS 从 EMPLOYEES 表中读取一行之后，在执行子查询时系统使用当前行（刚刚读入的行）中的 EMP_ID 列值。

在本例中的子查询的 WHERE 子句中引用的 EMP_ID 列是外引用——即引用的列名并不出现在子查询的 FROM 子句列出的表中。外引用是用了来自主查询的 FROM 子句列出的表中的列。另外，DBMS 从主查询的正在处理的当前行中提取外引用的值（在本例中是 EMP_ID）。

技巧 244 在子查询中使用 EXISTS 判式来决定行中是否有满足搜索标准的列值

EXISTS 判式允许检查一个表，看是否至少有一行满足一套搜索条件。实际上，EXISTS 总是与子查询结合使用，而且只要子查询至少返回一个值则求值为 TRUE。如果子查询的结果表中无值（意即表中没有行满足子查询的 WHERE 子句中的搜索条件），那么 EXISTS 判式求值为 FALSE。

例如，假设想要获得那些单个顾客的购货量超过 \$100,000 或是其顾客（总体看）总购货量超过 \$500,000 的销售人员的清单。换句话说，可以说：“请列出至少存在（EXISTS）一位顾

客其总定单多于\$100,000 或者存在 (EXISTS) 几个顾客的总购货量大于\$5000.00 的销售人员。”类似于下面的查询将生成所要的雇员清单：

```
SELECT emp_ID, f_name, l_name
FROM employees
WHERE EXISTS (SELECT * FROM customers
WHERE salesperson = emp_ID
AND (total_purchases > 100000
OR (SELECT SUM(total_purchases)
FROM customers
WHERE salesperson = emp_ID) > 500000))
```

为了执行查询，DBMS 对 EMPLOYEES 表一次处理一行。当在主查询中所传送的是正在处理的当前行的 EMP_ID 列值时，子查询生成一个中间（虚拟）表，其中有来自 CUSTOMERS 表中的一列的值，该列满足查询的 WHERE 子句中的搜索条件。

如果子查询执行后，中间（虚拟）表没有值，那么 EXISTS 判式求值为 FALSE，DBMS 转而处理 EMPLOYEES 表中的下一行。另一方面，如果子查询的中间（虚拟）表至少有一个值，EXISTS 求值为 TRUE，DBMS 在处理 EMPLOYEES 表中的下一行之前，将（在本例中）把雇员 ID、姓名添加到主查询的结果表中。

注意：一个子查询必须产生单值结果。给定一个有不止一列的 CUSTOMERS 表，EXISTS 测试中的子查询中的 SELECT * 似乎违反了这一规则。但是，由于 EXISTS 判式实际上并不使用由于查询生成的任何数据值（EXISTS 判式需要了解的只是是否生成了什么值），SQL 将子查询的 SELECT 子句中的 SELECT * 看作意义为“选择任何一列”，而不是“选择所有列”。

除了 EXISTS 判式之外（如果子查询产生至少一行则求值为 TRUE），还可使用 NOT EXISTS，此判式当子查询不产生行时求值为 TRUE。

技巧 245 使用关键词 IN 引入子查询

当使用关键词 IN 来引入子查询时，就告诉 DBMS 执行一种所谓的“子查询集成员测试”（subquery set membership test）。换句话说，就是告诉 DBMS 把单个数据值（放在关键词 IN 前面的）与由子查询（跟在关键词 IN 后的）生成的结果表中的列数据值相比较。如果单个数据值与由子查询返回的列数据值之一相匹配，那么 IN 判式求值为 TRUE。

根据子查询集成员测试用于生成结果表的句法如下：

```
SELECT <column name list> FROM <table list>
WHERE <test expression> IN <subquery>
```

其中<test expression>可以是实际值、列名、表达式或是另一个返回单个值的子查询。

例如，如果学生信息保存在 STUDENTS 表中，每个学生所选学的课程列在 ENROLLMENT 表中，那么可以用下面的 SELECT 语句来获得注册学习 ENGLISH-101 的学生的清单：

```
SELECT SID, f_name, l_name FROM students
WHERE SID IN (SELECT student_ID FROM enrollment
WHERE course_ID = 'ENGLISH-101')
```

在执行一个查询时，DBMS 首先处理最里面的子查询。这样一来，在本例中，系统首先生成 ENROLLMENT 表行中的那些在 COURSE_ID 列中有值为 ENGLISH-101 的行中的所有 STUDENT_ID 列的清单。接着，DBMS 一次处理 STUDENTS 表中的一行，并将每行中的 SID

列值与子查询的结果表中的学生 ID 相比较。

如果系统在子查询的结果表中找到了与正在处理的 STUDENTS 表行中的 SID 列值相匹配的 STUDENT_ID 列值，那么 WHERE 子句求值为 TRUE，系统在处理 STUDENTS 表中的下一行之前，将该学生的 ID 和姓名插入到主查询的结果表中。另一方面，如果 DBMS 没有在子查询的结果表中发现与 SID 的匹配，那么 WHERE 子句求值为 FALSE，DBMS 转而去处理 STUDENTS 表中的下一行，而不会在查询的结果表中插入该学生的 ID 和姓名。

NOT 关键词反转 IN 测试的效果。因此，如果用 NOT IN（而不是 IN）引入子查询，如果在子查询的结果表中没有匹配值，则 DBMS 采取 SQL 语句指定的行动。

注意：由关键词 IN 引入的子查询返回的列值既可来自用于主查询中的表之一，也可来自完全另外的表。SQL 对于查询的惟一要求是，它必须返回单一列的数据值，其数据类型要与关键词 IN 前的表达式的数据类型兼容。

技巧 246 使用 ALL 引入返回多个值的子查询

只要使用 6 种 SQL 比较运算符（=、<>、>、>=、<或<=）中的一个来比较两个表达式的值，那么运算符前后的表达式都必须求值为单一值。因而，仅当子查询产生用于比较测试的单值时，才可使用一个子查询作为比较判式中的表达式之一。

例如，如果有一个 AUTO_POLICIES 表和一个 ACCIDENTS 表，可提交如下查询让 DBMS 生成车速在每小时 35 英里（MPH）或以下至少发生过一次翻车事故的保单的清单：

```
SELECT policy_number, make, model FROM auto_policies p
WHERE (SELECT COUNT(*) FROM accidents a
WHERE (description LIKE '%rollover%' AND mph <= 35)
AND a.make = p.make
AND a.model = p.model) > 0
```

如果 ACCIDENTS 表中有某种在主查询中正在测试的牌号和车型的有关车速在 35MPH 或以下的有不止一行 ROLLOVER 细节行，子查询将生成多于一行的结果表。但是，该子查询仍然是比较判式中的合法表达式，因为 COUNT(*) 总计函数将子查询生成的结果表减少为单值。

现在，假设想要得到对车速在 35 MPH 或以下时有最高的翻车机会的汽车牌号和车型的保单清单。由总计函数为整个表返回的单值，如 COUNT(*)，在这类查询中还不够。此时需要 DBMS 分类汇总由某一牌号和车型的汽车发生的翻车次数，然后需要系统将每种牌号和车型的翻车次数与其他牌号的车型的最大翻车次数加以比较。

数量词 ALL、SOME 和 ANY 允许使用比较运算符将单值与子查询返回的值加以比较。例如，下面的查询将列出 POLICIES 表中那些比 ACCIDENTS 表中其他牌号和车型的翻车次数要多的牌号和车型：

```
SELECT policy_number, f_name, l_name, make, model
FROM auto_policies p
WHERE (SELECT COUNT(*) FROM accidents a
WHERE (description LIKE '%rollover%' AND mph <= 35)
AND a.make = p.make
AND a.model = p.model)
> ALL
(SELECT COUNT(*) FROM accidents a
```

```
WHERE (description LIKE '%rollover%' AND mph <= 35)
AND (a.make <> p.make OR a.model <> p.model)
GROUP BY make, model)
```

第一个子查询（在> ALL 比较的左边）返回单值——即在主查询中正在测试的牌号和车型所发生的翻车事故的数目。同时，第二个子查询（在> ALL 比较的右边）返回 ACCIDENTS 表中每种其他牌号和车型的翻车事故的次数。

虽然大于比较运算符(>)右边的子查询有不只一个值，ALL 量词允许将其用在比较中，来告诉 DBMS 将有单值的值清单中的多个值，一次比较一个值。换句话说，ALL 量词将比较从单值与多值集合的比较改变为多个单值的比较——对于子查询的结果表中的每行进行一次。

如果所有(all)比较测试都求值为 TRUE，那么 WHERE 子句求值为 TRUE，主查询就将来自 AUTO_POLICIES 表中的当前行的列值添加到 SELECT 语句的结果表中。另一方面，如果有任何一个比较求值为 FALSE，那么 WHERE 子句求值为 FALSE，DBMS 转而处理 AUTO_POLICIES 表中的下一行，而不向查询的最后结果表中添加任何列数据值。

技巧 247 在子查询中使用总计函数返回单值

除了在查询的结果表中包括总计函数结果外，还可以作为子查询的 WHERE 子句的搜索条件的一部分得到总计函数返回的值。例如，如果想要显示 PRODUCTS 表中的最高价格货品的 PRODUCT_ID、DESCRIPTION 和 ITEM_COST 时，可提交如下查询：

```
SELECT product_id, description, item_cost
FROM products
WHERE item_cost = (SELECT MAX(item_cost) FROM products)
```

由于总计函数总是返回单个值，因而可将 SELECT 子句中只有一个总计函数的子查询作为比较判式两边的任何一个表达式。毕竟，在 SELECT 子句中只有一列的子查询可保证总是返回一个且只有一个值，即总计函数返回的单个值。因而，为了获得那些平均总定单大于总体定单总数的雇员的清单，可提交以下查询：

```
SELECT emp_id, f_name, l_name FROM employees e
WHERE (SELECT AVG(order_total) FROM orders o
WHERE o.salesperson = e.emp_id) >
(SELECT AVG(order_total) FROM orders)
```

请注意，两个子查询中的每个中的 SELECT 子句包括单个总计函数 AVG(ORDER_TOTAL)。因此，每个子查询是 DBMS 可使用 6 种比较运算符（如本例中的>）在比较判式中加以比较的单值表达式。

技巧 248 理解 WHERE 子句中子查询的作用

WHERE 子句中的子查询就像是用于包括查询（或子查询）的行选择过程的过滤器。例如，如果想要了解高于平均 GROSS_SALES_YTD 的销售人员的姓名，可执行如下查询：

```
SELECT f_name, l_name FROM employees
WHERE dept = 'Sales'
AND gross_sales_ytd > (SELECT AVG(gross_sales_ytd)
FROM employees WHERE dept='Sales')
```

DBMS 执行内层查询得到销售部门内所有雇员的平均的 GROSS_SALES_YTD。接着用这

个数字从 EMPLOYEES 表中过滤掉那些 GROSS_SALES_YTD 值小于平均值（由子查询中的 AVG 总计函数返回的）的那些雇员。

在这个简单的示例中，可对表本身执行子查询以得到平均的 GROSS_SALES_YTD。然后，当知道了平均值后，可将其用来代替 SELECT 语句中的子查询。例如，假设 GROSS_SALES_YTD 列中的平均值为 \$227,500.50。可将查询改写为：

```
SELECT f_name, l_name FROM employees
WHERE dept = 'Sales' AND gross_sales_ytd > 227500.50
```

但是，如果将 GROSS_SALES_YTD 数字硬性编码到查询中，则当下次想要列出高于平均的销售人员时，必须再次执行（子）查询并改写查询。子查询让人们根据随时间改变的值过滤掉不想要的行，而不必事先了解“过滤器”的值，从而避免此类重复的工作。

另外，当有大量要执行的计算时，使用 WHERE 子句中的子查询来过滤掉不想要的行也比编写并执行多个查询省很多时间。例如，假设 EMPLOYEES 表中没有 GROSS_SALES_YTD 列。为了获得每个销售人员的毛销售数字，必须为每个销售人员的顾客定单计算 ORDERS 表的 ORDER_TOTAL 列中的值的总和。如果公司有大量的销售人员，则重复地执行如下查询（在本例中，即用每个销售人员的 EMP_ID 来代替查询中的 11），以便获得每个雇员的总销售额：

```
SELECT SUM(order_total) FROM orders WHERE salesperson = 11
```

这可能是非常费时的事，特别是当必须按月、按周或甚至按日来完成报告时更是如此。

如果在 WHERE 子句中使用子查询（或多个子查询），可让 DBMS 计算每个销售员的平均定单（YTD），决定所有销售人员的平均销售额（YTD）并报告那些销售业绩高于平均的销售人员的姓名，所有这些运算都可在下面的一个查询中完成：

```
SELECT f_name, l_name FROM employees
WHERE dept = 'Sales'
AND (SELECT AVG(order_total) FROM orders
WHERE salesperson = emp_ID
AND order_date >= '01/01/2000')
> (SELECT AVG(order_total) FROM orders
WHERE order_date >= '01/01/2000')
```

要理解的重要事情是，WHERE 子句中的子查询可用于计算 DBMS 用在搜索条件中的值（或值集）——就像任何其他列引用的实际值一样。正如通常情况一样，DBMS 只将那些来自（联合）行中的 WHERE 子句的搜索条件求值为 TRUE 的行添加到最后的结果表中。

技巧 249 使用嵌套查询返回 TRUE 或 FALSE 值

在技巧 248 “理解 WHERE 子句中子查询的作用”中，读者了解到，可将子查询用于计算在 WHERE 子句中与比较运算符一起使用的值。但有时想要根据一个表中是否有满足搜索条件的行，而不是看列或列集是否有指定的值来采取行动。使用关键词 EXISTS 引入的 WHERE 子句子查询将返回简单的 TRUE 或 FALSE，DBMS 可根据这样的值来决定是否采取由 SQL 语句指定的行动。

例如，如果保险公司想要对在过去的一年中曾经赔付过的小轿车保单追加 10% 的额外保费，可提交如下查询：

```
SELECT policy_number, RTRIM(f_name)+' '+l_name AS 'Name',
cost * .10 AS 'Surcharge'
```

```
FROM auto_policies ap
WHERE EXISTS (SELECT * FROM claims_paid c
              WHERE c.policy_number = ap.policy_number
              AND claim_date > GETDATE() - 365)
```

此查询将对列在 CLAIMS_PAID 表中的每个保单列出 POLICY_NUMBER、投保者的姓名和 SURCHARGE。作为对照，如果子查询找不到来自主查询的 AUTO_POLICIES 表的 POLICY_NUMBER 列，则 EXISTS 判式返回 FALSE，系统就不把来自正在处理的当前行中的列数据值添加到最后的结果表中。

类似地，当子查询中的表没有满足 WHERE 子句搜索条件的行时，由 NOT EXISTS 引入的子查询将返回布尔值 TRUE。例如，假设前例中的 AUTO_POLICIES 表已改变为包括 SURCHARGE 和 DISCOUNT 列。如果想要对公司曾经赔付过的保单在 SURCHARGE 列记录下 10% 的额外保费，可使用以下的 UPDATE 语句：

```
UPDATE auto_policies
SET surcharge = .1 * cost, discount = 0.00
WHERE EXISTS (SELECT * FROM claims_paid c
              WHERE c.policy_number =
                    auto_policies.policy_number
              AND claim_date > GETDATE() - 365)
```

类似地，为了对那些在过去一年中没有赔付过的保单记录下 10% 的折扣，可执行如下的 UPDATE 语句：

```
UPDATE auto_policies ap
SET discount = .1 * cost, discount = 0.00
WHERE NOT EXISTS (SELECT * FROM claims_paid c
                  WHERE c.policy_number =
                        auto_policies.policy_number
                  AND claim_date > GETDATE() - 365)
```

当 DBMS 执行本例中的第二个 UPDATE 语句时，如果其子查询没有返回行，则 NOT EXISTS 判式将返回 TRUE。因而，如果子查询中的 ACCIDENTS 表中没有与来自主查询的 AUTO_POLICIES 表中正在处理的当前行中的 POLICY_NUMBER 相同的行，那么子查询结果表中将不包括行。EXISTS 求值为 FALSE。但是，NOT 关键词将反转 EXISTS 的结果值，因而 NOT EXISTS 判式（及整个 WHERE 子句）求值为 TRUE。当本例中的第二个 UPDATE 语句中的 WHERE 子句求值为 TRUE 时，DBMS 将把 DISCOUNT 列值设置为保单价值的 10%，还把 SURCHARGE 列值设置为 0。

技巧 250 理解 HAVING 子句中子查询的作用

在技巧 248 “理解 WHERE 子句中子查询的作用”中，读者了解到，可使用 WHERE 子句中的子查询过滤掉单独的不想要的行。HAVING 子句中的子查询也可用于过滤过程。但是，不是使用 HAVING 子句中的子查询的结果来过滤掉单独的不想要的行，而是用子查询的结果表一次过滤掉一组或几组行。

虽然 WHERE 子句既可出现在组合查询中也可出现在非组合查询中，但 HAVING 子句（如果存在的话）几乎总是跟随组合查询的 GROUP BY 子句。例如，以下组合查询中的 HAVING

子句:

```
SELECT f_name, l_name, SUM(order_total) AS 'Total Orders'
FROM employees, orders
WHERE employees.emp_ID = orders.salesperson
AND order_date BETWEEN '10/01/2000' AND '10/31/2000'
GROUP BY f_name, l_name
HAVING SUM(order_total) > .25 *
(SELECT SUM(order_total) FROM orders
WHERE order_date BETWEEN '10/01/2000' AND '10/31/2000')
```

告诉 DBMS 列出那些在 2000 年 10 月的总销售额代表了该月的所有销售人员的销售总额的 25% 的销售人员。由于 HAVING 子句中的子查询没有外部列引用, DBMS 可把总定单额乘以 .25 一次, 然后, 当 HAVING 子句测试每个行组时, 重复地使用该乘积 (作为常数) 以获得最后的结果表 (毕竟, 2000 年 10 月的 ORDER_TOTAL 的 25% 对于 DBMS 将其与由第 1 个、第 10 个或第 n 个销售人员所售出的定单组的比较来说, ORDER_TOTAL 是不变的)。

在执行主查询时, 系统一次处理联合 (EMPLOYEES/ORDERS) 行中的一行, 根据销售人员将定单组合 (总和) 在一起。然后 DBMS 使用 HAVING 子句中的搜索条件过滤掉不想要的行组。在本例中, 系统删除那些在组中 ORDER TOTAL 列值的总和小于或等于 2000 年 10 月中所有 ORDER_TOTAL 列值的汇总的 25% 的组。

除了非关联子查询 (也就是没有外引用的子查询) 之外, 还可在 HAVING 子句中使用关联子查询。例如, 假设想要列出那些在 2000 年 10 月的销售额超过前月个人总销售额的 50% 的销售人员的姓名和总销售额的清单。由于子查询包括外引用 (在主查询中正在处理的行组中的 EMP_ID 列的引用), DBMS 必须多次执行查询的 HAVING 子句中的关联子查询 (对于在 2000 年 10 月中有销售额的每个销售人员执行一次):

```
SELECT f_name, l_name, SUM(order_total) AS 'Total Orders'
FROM employees, orders, emp_ID
WHERE employees.emp_ID = orders.salesperson
AND order_date BETWEEN '10/01/2000' AND '10/31/2000'
GROUP BY f_name, l_name, emp_ID
HAVING SUM(order_total) >= 1.5 *
(SELECT SUM(order_total) FROM orders
WHERE salesperson = emp_ID
AND order_date BETWEEN '09/01/2000' AND '09/30/2000')
```

为了执行本例中的 SELECT 语句, DBMS 一次处理 EMPLOYEES/ORDERS 表中的一组联合行, 总计销售人员的定单。不是使用常数值 (在应用 HAVING 子句过滤器之前计算一次), DBMS 必须在处理来自下一销售人员之前计算一个销售人员在 2000 年 9 月中的总销售量。如果正在测试的行组中的 ORDER_TOTAL 列值的和不超过子查询结果表中的值的 50%, DBMS 就从结果表过滤掉这一行组。

现在要理解的重要事情是, HAVING 子句中的子查询可用于与 WHERE 子句中的子查询同样的目的。在执行子查询一次或多次之后 (如果 HAVING 子句有一个关联的子查询, 则对正在处理的每组或一些行执行一次), 系统使用子查询返回的值从包括查询的结果表中过滤掉不想要的 (联合) 行。

技巧 251 理解 JOIN 语句中关联的和非关联的子查询的执行顺序

引用了来自包括（或外部）查询的子查询称为关联子查询，因为子查询的结果与包括（外部）查询中的每一行都是相关的。

当执行有非关联子查询的 JOIN 语句时，DBMS 首先将主查询表中的相关行联合起来。接着，系统使用主查询的 WHERE 子句中的任何非子查询的搜索条件来过滤掉不想要的行。然后 DBMS 执行子查询从中间（虚拟）结果表中过滤掉其余不想要的行。最后，DBMS 将来自余下的联合行的列值发送到 SELECT 子句中，用于查询的结果表输出。

例如，假设为了生成在 2000 年中购买超过 \$100,000 货品的每个账户的顾客和销售人员的清单，提交如下的 JOIN 语句：

```
SELECT RTRIM(c.f_name)+' '+c.l_name AS 'Customer',  
RTRIM(e.f_name)+' '+e.l_name AS 'Salesperson'  
FROM customers c JOIN employees e  
ON salesperson = emp_ID  
WHERE cust_ID IN  
(SELECT cust_ID FROM orders  
WHERE order_date BETWEEN '01/01/2000' AND '12/31/2000'  
GROUP BY cust_ID  
HAVING SUM(order_total) > 100000.00)
```

DBMS 首先生成 CUSTOMERS 和 EMPLOYEES 表的 CROSS JOIN。接着使用 WHERE 子句中的非子查询条件过滤掉那些 SALESPERSON 列值（来自 CUSTOMERS 表）与 EMP_ID（来自 EMPLOYEES 表）不匹配的行。系统还要过滤掉定单日期早于 01/01/2000 或晚于 12/31/2000 的所有行。然后 DBMS 执行子查询生成来自定单总量大于 \$100,000.00 的顾客的 CUST_ID 值的表。最后，系统一次处理中间主查询表中的一行，检查由子查询返回的 CUST_ID 值的清单中是否存在联合行中的 CUST_ID。如果来自联合行中的 CUST_ID 存在于由子查询返回的清单中，DBMS 将顾客及雇员姓名列值发送到 SELECT 子句中，用于最后的结果表输出。

如果 JOIN 语句中的子查询是非关联的子查询，也就是说，该子查询没有对包括查询的列值的外引用，DBMS 只需要执行该子查询一次。例如，在本例中，DBMS 只需要生成 CUST_ID 清单一次，因为不管 DBMS 是测试第一行还是测试第 200 行，购买超过 \$100,000.000 产品的顾客清单都是一样的。但是，如果 JOIN 语句使用了关联子查询，DBMS 必须重复地执行该子查询，对还没有过滤掉的中间结果表中的每一行都要执行一次。

例如，如果将前面示例中的查询改写如下：

```
SELECT RTRIM(c.f_name)+' '+c.l_name AS 'Customer',  
RTRIM(e.f_name)+' '+e.l_name AS 'Salesperson'  
FROM customers c JOIN employees e  
ON salesperson = emp_ID  
WHERE (SELECT SUM(order_total) FROM orders o  
WHERE o.cust_ID = c.cust_ID  
AND order_date  
BETWEEN '01/01/2000' AND '12/31/2000')  
> 100000.00
```

其中 WHERE 子句中的子查询是关联的，因为它引用了包括查询中的 CUST_ID 列值。由

于每个顾客定单的总和是不一样的，DBMS 必须重复地执行子查询，对正在测试的中间表（CUSTOMERS/EMPLOYEES）中的每个联合行都要执行一次，以便获得最后的结果表。

技巧 252 使用关键词 IN 引入关联子查询来确定有特定值的表列的存在性

关联的子查询可用在任何非关联子查询可以出现的 SQL 语句中。例如，下面的查询将显示 CUSTOMERS 表中那些在 SHIP_TO_STATE 列中有值为 NV 的行：

```
SELECT cust_ID, f_name, l_name, street_address, state,
       zip_code
FROM customers
WHERE 'NV' IN (SELECT ship_to_state FROM orders o
               WHERE o.cust_ID = c.cust_ID)
```

注意：由于关联子查询的较重的处理需求，最好只在没有替代方法时才使用这种子查询。在本例中，DBMS 将重复执行子查询，生成那些对于每个顾客定单被发送一次的州的清单。更为有效的替代方法是，既可将本例中的查询改写为 JOIN 语句：

```
SELECT DISTINCT c.cust_ID, f_name, l_name, street_address,
               state, zip_code
FROM customers c JOIN orders o
ON o.cust_ID = c.cust_ID
WHERE ship_to_state = 'NV'
```

也可改写为如下的带 WHERE 子句的多表 SELECT 语句：

```
SELECT DISTINCT cust_ID, f_name, l_name, street_address,
               state, zip_code
FROM customers c, orders o
WHERE c.cust_ID = o.cust_ID
AND ship_to_state = 'NV'
```

技巧 253 理解用比较运算符引入的关联子查询

与非关联子查询的情况一样，可将关联子查询用在几乎凡是表达式可以出现的任何地方。但是，根据子查询所使用的上下文环境，SQL 对子查询的结果施加了某些限制。例如，如果将关联子查询用作比较运算符（=、<、>、>=、<或<=）之后的表达式中，该子查询必须返回单值。

例如，假设可根据过去 30 天中的购货量给予顾客的下次定单多达 6% 的折扣，折扣多少安排如下表所示：

lower_limit	upper_limit	discount
0.00	9,999.99	0%
10,000.00	19,999.99	1%
20,000.00	29,999.99	2%
30,000.00	39,999.99	3%
40,000.00	49,999.99	4%
50,000.00	59,999.99	5%
60,000.00	999,999.99	6%

则以下查询将显示那些在过去 30 天至少有一个定单的顾客的 CUST_ID、姓名和折扣百分数：

```

SELECT cust_ID, RTRIM(f_name)+' '+l_name AS 'Customer',
(SELECT discount FROM discount_schedule
WHERE lower_limit <= (SELECT SUM(order_total) FROM orders
WHERE orders.cust_ID = c.cust_ID
AND order_date >= GETDATE() - 30
AND order_date < GETDATE()))
AND upper_limit >= (SELECT SUM(order_total) FROM orders
WHERE orders.cust_ID = c.cust_ID
AND order_date >= GETDATE() - 30
AND order_date < GETDATE()))
AS 'Pet_Discount'
FROM customers c
WHERE c.cust_ID IN (SELECT cust_ID FROM orders
WHERE order_date >= GETDATE() - 30
AND order_date < GETDATE())

```

在本例中，由两个比较运算符（<= 和 >=）中的每个引入的关联子查询产生单值——即由主查询的正在处理的行中 CUST_ID 表示的顾客所下定单的 ORDER_TOTAL 列的总和。由于关联子查询引用包括查询正在测试的行中的列，则 DBMS 必须重复地执行子查询。因而，在本例中，系统每当测试另一顾客行（来自 CUSTOMERS 表）时，都将总和 orders 表中的 ORDER_TOTAL 列，以便获得最后的结果表。

DBMS 优化器应该明了以下事实，两个关联子查询是一样的，而且应该只总和每个顾客的 ORDER_TOTAL 列一次。查询可改写成另外一种形式：

```

SELECT cust_ID, RTRIM(f_name)+' '+l_name AS 'Customer',
(SELECT discount FROM discount_schedule
WHERE (SELECT SUM(order_total) FROM orders
WHERE orders.cust_ID = c.cust_ID
AND order_date >= GETDATE() - 30
AND order_date < GETDATE())
BETWEEN lower_limit AND upper_limit) AS 'Pet_Discount'
FROM customers c
WHERE c.cust_ID IN (SELECT cust_ID FROM orders
WHERE order_date >= GETDATE() - 30
AND order_date < GETDATE())

```

这里只有一个关联子查询。关键词 BETWEEN 对 DISCOUNT_SCHEDULE 表中的 LOWER_LIMIT 和 UPPER_LIMIT 列值执行大于、等于（>=）和小于或等于（<=）测试。

注意：本例中的关联子查询本身是另一子查询（将顾客的折扣百分数放入主查询最后的结果表中的 PCT_DISCOUNT 列）的子查询。请注意，关联子查询可对包括该子查询的查询进行外部列引用。在本例中，关联子查询不是对包括查询进行外部列引用，而是对主查询（它包括了子查询和该子查询的关联子查询）正在测试的行的 CUST_ID 列作出了列引用。

技巧 254 将关联子查询用作 HAVING 子句中的过滤器

虽然 DBMS 使用 WHERE 子句中的搜索条件来过滤掉查询的结果表中单独的不想要的行，但系统使用 HAVING 子句中的搜索条件删除那些搜索条件求值为 FALSE 的行组。在技巧 250

“理解 HAVING 子句中子查询的作用”中，读者已经了解到，子查询可用作 HAVING 子句的搜索条件中的表达式之一。当把关联子查询也用作 HAVING 子句的搜索条件中的表达式之一时，人们应该不会惊奇。

例如，假设想要看一看一个组织在管理级别上是否有些“超重”，因为其中任何一个部门经理都要比工作人员多。类似如下的查询，将显示那些职位为经理(manger)、经理助理(assistant manger)、主管(supervisor)、副总经理(vice president)或总经理(president)的雇员人员比不是这些职位的雇员人数多的部门的名称：

```
SELECT dept FROM employees e
GROUP BY dept
HAVING (SELECT COUNT(*) FROM employees cs_e
WHERE e.dept = cs_e.dept
AND(position LIKE '%Manager%' OR
position LIKE '%Supervisor%' OR
position LIKE '%President%')) >
(SELECT COUNT(*) FROM employees cs_e
WHERE e.dept = cs_e.dept
AND(position NOT LIKE '%Manager%' AND
position NOT LIKE '%Supervisor%' AND
position NOT LIKE '%President%'))
ORDER BY dept
```

在执行这个 SELECT 语句时，DBMS 首先根据部门在中间表中组合来自 EMPLOYEES 表中的所有行，在概念上如图 13.1 所示。

EMP_ID	F_NAME	L_NAME	DEPT	POSITION
13	Clint	Rutberg	HR	Manager
14	Linda	Eastwood	HR	Clerk
5	Sally	Lauren	Marketing	Floor Supervisor
6	Hillary	Rafer	Marketing	Floor Supervisor
7	Jessica	Lewinski	Marketing	Day Manager
8	Chris	Hahn	Marketing	Telemarketer
9	Walter	Mathews	Marketing	Telemarketer
10	James	Mondale	Marketing	Night Manager
11	Clide	Dean	Marketing	Floor Supervisor
15	George	Tyson	Office	Manager
16	Michele	Harrison	Office	Receptionist
⋮	⋮	⋮	⋮	⋮

图 13.1 根据 DEPT 列组合的组合查询中间（虚拟）EMPLOYEES 表中的行

接着，系统一次处理虚拟表中的一个部门，对每个部门执行 HAVING 子句中的关联子查询，并使用子查询的结果过滤掉那些“管理人员”数小于或等于“工作人员”数的部门的行组。最后，DBMS 将来自每个余下的组中的 DEPT 列发送到 SELECT 子句，以备在最后的結果表中输出。

技巧 255 使用关联子查询为 UPDATE 语句选择行

如果提交一条没有 WHERE 子句的 UPDATE 语句, DBMS 将改变整个表中保存在 UPDATE 语句的 SET 子句指定的列中的值。例如, 如果提交以下 UPDATE 语句, DBMS 将把存货目录中的每项货品的 PRICE 列值减少 25%:

```
UPDATE inventory SET price = price * .75
```

虽然这会使销售人员和顾客非常高兴, 但要回答股票持有人为什么要对那些已经短缺的货品还要减少公司的利润的问题。

为了限定所做的变化, 通过向 UPDATE 语句添加 WHERE, 将对表中的特定行做出改变。例如, 只对那些在过去 30 天中没有卖出去的货品打折, 可将前例中的 UPDATE 语句改写如下:

```
UPDATE inventory SET price = price * .75
WHERE NOT EXISTS
  (SELECT * FROM invoice_details i
   WHERE inventory.item_no = i.item_number
   AND inv_no IN
    (SELECT inv_no FROM invoices
     WHERE inv_date >= GETDATE() - 30))
```

当处理修改后的 UPDATE 语句时, DBMS 首先执行子查询 (WHERE 子句中的关联子查询内的非关联子查询), 从而生成在过去 30 天中卖出过货的发票号的清单。系统一次处理 INVENTORY 表中的一行 (一种货品), 通过对 INVENTORY 表中的每项货品号 (ITEM_NO) 执行关联子查询, 测试是否需要更新 (UPDATE) 该行。

每当执行时, 关联子查询根据在过去 30 天内输入的发票建立起 INVOICE_DETAILS 行的清单, 其中 ITEM_NUMBER (来自 INVOICE_DETAILS 表) 与 INVENTORY 表中正在测试的行中的 ITEM_NO 相匹配。如果货品号未包括在过去 30 天内生成的任何发票明细记录 (行) 中, 那么子查询不返回行——这就使得 EXISTS 子句求值为 FALSE。但是, 关键词 NOT (它引入了 EXISTS 子句) 将 FALSE 改变为 TRUE, 从而使得 WHERE 求值为 TRUE。因此, WHERE 子句使得 DBMS 把那些在过去 30 之内没有销售发票的货品价格减少 25%。

要理解的重要事情是, UPDATE 语句的关联子查询中的外部列引用允许实现子查询与 UPDATE 目标表之间的 JOIN。虽然 SQL-89 规范禁止 UPDATE 语句中的子查询对正在更新的表中的列进行列引用, 但 SQL-92 规范没有这一限制。由此, 以下 UPDATE 语句在 SQL-89 规范中是禁止的, 但在 SQL-92 规范中却完全是合法的 (此语句减免了所有超过 \$1,000.00 定单的运输和处理费用):

```
UPDATE orders SET shipping_and_handling = 0
WHERE (SELECT SUM(order_total) FROM orders o
      WHERE o.order_number = orders.order_number)
> 1000.00
```

注意: 如果 UPDATE 语句的关联子查询对语句本身正在改变的列进行外部列引用, DBMS 在整个 UPDATE 语句执行过程中使用列中变化前的值。

第 14 章 理解事务处理隔离级别和并发处理

技巧 256 使用带关联子查询的 INSERT 语句创建快照表

可在 INSERT 语句中使用子查询，不仅可返回多个行，而且还可返回多个数据列（使用 IN、ANY、SOME 或 ALL 引入每种子查询时，子查询可返回多行，但只有单列）。因而，为了把一个表中的全部内容传送到另一表中，可执行如下的 INSERT 语句：

```
INSERT INTO table2 (SELECT * FROM table1)
```

只要 TABLE1 和 TABLE2 有完全一样的结果结构（同样的数据类型、同样的列数，并以同样的顺序排列）。

如果想要只将来自 TABLE1 中的某些列中的数据复制到 TABLE2 中，只要修改 INSERT 语句和子查询，使之包括所要复制的列的清单即可。例如，假设想要根据保存在 ITEM_MASTER 表中的数据为销售人员生成一个 PRICE_LIST 表。带子查询的如下 INSERT 语句将从 ITEM_MASTER 表中复制 ITEM_NUMBER 和 DESCRIPTION 列并对保存在 PRICE_LIST 表的 PRICE（而不是保存在 ITEM_MASTER 表中的实际 COST）建立 20% 的利润空间：

```
INSERT INTO price_list (product_code, description, price)
(SELECT item_number, description, ROUND(cost * 1.20, 2)
FROM item_master)
```

顾名思义，快照（snapshot）表是另一表在特定时刻的数据的副本（或图景）。例如，假设想要保存月底存货目录的快照（有了月底的存货目录的副本将可对特定日期上的诸如总价值、货品数量以及一月到另一月的货品数量的比较等写出报告，而不必翻译顾客所下的全部定单）。例如，如果创建了一个与存货目录表（INVENTORY）有相同结构的 INVENTORY_10_31_2000 这样的表，可提交如下的 INSERT 语句将 10/31/2000 那天的 INVENTORY 表保存为 INVENTORY_10_31_2000 表：

```
INSERT INTO inventory_10_31_2000 (SELECT * FROM inventory)
```

如果想要对表中的部分数据提取快照，只要在子查询中添加 WHERE 子句即可。例如，为了将 ORDERS 在 2000 年 10 月的所有定单记录的快照保存起来，可提交如下的 INSERT 语句：

```
INSERT INTO orders_Oct_2000
(SELECT * FROM orders
WHERE order_date BETWEEN '10/01/2000' AND '10/31/2000')
```

或者，为了只提取产品 XYZ 在 2000 年 10 月期间的定单的快照，可提交如下的 INSERT 语句：

```
INSERT INTO XYZ_orders_Oct_2000
(SELECT * FROM orders
WHERE order_date BETWEEN '10/01/2000' AND '10/31/2000'
AND product_code = 'XYZ')
```

技巧 257 GRANT 语句授予某人以 DELETE 权限

作为一名数据库对象拥有者 (DBOO)，对所创建的数据库对象有所有的权限 (INSERT、UPDATE 和 DELETE)。另外可向其他用户授予 (GRANT) 这些权限中的任何一种或所有权限。因而，如果某人创建了 EMPLOYEES 表和 TIMECARDS 表，或许要给予人事 (人力资源) 部门的某些人以删除 (DELETE) 雇员信息和考勤行的能力 (当雇员离开了公司时)。类似地，如果创建了一个 CUSTOMERS 表，可能会向销售经理授予 DELETE 权限，以删除那些在一段时间以来没有购过货的顾客。

GRANT DELETE 语句的句法如下：

```
GRANT DELETE ON <table name>|<view_name> TO <username|role>
[...,<last username|role>][WITH GRANT OPTION]
```

这样一来，为了允许用户名 MARY、SUE 和 FRANK 可以从 EMPLOYEES 表中删除行，可执行以下的 GRANT DELETE 语句：

```
GRANT DELETE ON employees TO mary, sue, frank
```

类似地，如果用户 TOM、HELEN、SUSAN 和 RODGER 是 PERSONNEL 角色 (组) 中的成员，可使用如下 GRANT DELETE 语句向他们全体授予对 TIMECARDS 表的 DELETE 权限：

```
GRANT DELETE ON timecards TO personnel
```

读者已在技巧 101、技巧 104 和技巧 105 中学习了有关组 (即数据库角色) 安全性的内容。

如果在 GRANT DELETE 语句中包括 WITH GRANT OPTION，可允许被授予 DELETE 权限的用户再向其他用户授予此项权限。因此，无论何时在使用 WITH GRANT OPTION 时一定要小心。如果这样做，必须信任被授予 DELETE 权限的人，不仅允许其删除不再需要的那些表，而且还在向其他用户授予 DELETE 权限时也要像自己一样地表明合理性。因而，例如，如果 SUE 和 BILL 是销售部门的经理，可允许其中两个以及他们选定的另外一个人从 CUSTOMERS 表中删除行，可执行以下的 GRANT DELETE 语句：

```
GRANT DELETE ON customers TO sue, bill WITH GRANT OPTION
```

注意：当使用 GRANT DELETE 语句向一位用户授予 DELETE 权限时，还应该使用 GRANT SELECT 语句。如果只使用 GRANT DELETE 语句，那么该用户不能从表中选择所要删除的行。例如，如果只使用 GRANT DELETE 语句向用户 SUE 授予对 TIMECARDS 的 DELETE 权限，那么 SUE 可执行以下的 DELETE 语句：

```
DELETE timecards
```

这就删除了 TIMECARDS 文件中的所有行。但是，如果 SUE 想要通过执行以下的 DELETE 语句只删除那些 4 年以上的考勤卡：

```
DELETE timecards WHERE card_date < GETDATE() - 1460
```

此时 DBMS 将中止此 DELETE 语句的执行并显示如下错误消息：

```
Server: Msg229, level 14, State 5, Line 1
```

```
SELECT permission denied on object 'timecards', database
'SQLTips', owner 'dbo'.
```

因而，由于很少 (如果有的话) 想让一位被授予 DELETE 权限的用户去删除表中的所有行，一定要既授予 DELETE 权限还要授予 SELECT 权限，才能让用户能够删除表中的特定的行。

技巧 258 理解（CASCADE 和非 CASCADE）取消 GRANT 权限的效果

正如读者在技巧 107 “理解 REVOKE 语句”中所学习的，REVOKE 语句允许删除以前向某一用户所授予的权限。另外，由于授予权限本身也是一种权限，可使用以下的 REVOKE 语句的句法取消用户向其他人授予权限的权限，而原有的权限不变：

```
REVOKE [GRANT OPTION FOR] <privilege list> ON <object>
FROM <username|role>[..., <last username|role>]
[RESTRICT|CASCADE]
```

例如，如果执行以下 GRANT 语句，用户 SUE 将获得向 EMPLOYEES 表中插入新行的权限，同时也获得了向其他用户授予对 employees 表的 INSERT 权限的权限：

```
GRANT INSERT ON employees TO sue WITH GRANT OPTION
```

如果 SUE 执行下面的 GRANT 语句：

```
GRANT INSERT ON employees TO frank, mary WITH GRANT OPTION
```

则 FRANK 和 MARY 也能够向 EMPLOYEES 表中插入行。另外，所有这些用户都可以向其他用户授予对 EMPLOYEES 表的 INSERT 权限。

如果今后决定撤销（REVOKE）SUE 向其他用户的对 EMPLOYEES 表的 GRANT INSERT 权限，为达此目的，既可使用带 CASCADE 选项也可不带 CASCADE 选项的 REVOKE 语句。通过执行下面的 REVOKE 语句：

```
REVOKE GRANT OPTION FOR INSERT ON employees FROM sue
```

此时仍然允许 SUE、那些由 SUE 授予 INSERT 权限的用户（在本例中是 FRANK 和 MARY）以及任何被授予 INSERT 权限的用户（因为 SUE 是用 WITH GRANT OPTION 选项向别人授予对 EMPLOYEES 表的权限的）继续向 EMPLOYEES 表中插入行。另一方面，如果在执行 REVOKE 语句时包括了 CASCADE 选项，如下所示：

```
REVOKE GRANT OPTION FOR INSERT ON employees FROM sue
CASCADE
```

SUE 仍然能够向 EMPLOYEES 表中插入行。但是 SUE 以及原来被授予对 EMPLOYEES 表的 INSERT 权限的用户都不能再向其他用户授予对 EMPLOYEES 表的 INSERT 权限——除非这些人还从别人那里接受了这种权限。因而，在执行了本例中的 REVOKE 语句之后，SUE 还能向 EMPLOYEES 表中插入行，但 FRANK 和 MARY 就不再能这样做了，因为他们是从 SUE 的 GRANT INSERT 权限处接受的向 EMPLOYEES 表插入行的权限（而这种权限现在已经撤销了）。

注意：某些 DBMS 产品，如 MS-SQL Server，不允许在不包括 CASCADE 选项的情况下撤销（REVOKE）权限或是授予（GRANT OPTION）权限。如果要试图这样做，MS-SQL Server 将以下面的错误消息中止 REVOKE 语句的执行：

```
Server: Msg 4611, Level 16, State 1, Line 1
To revoke grantable privileges, specify the CASCADE option
with REVOKE.
```

因此，一定要检查系统手册，看一下具体的 DBMS 产品是否允许在执行 REVOKE 语句删除以前用 GRANT OPTION 授予的权限时忽略 CASCADE 选项。

技巧 259 理解如何一起使用 GRANT 和 REVOKE 语句以便在授予权限时节省时间

当必须对一个表（或几个表）中的多列向几个用户授予权限时，有时向所有用户授予较大范围的权限，然后再撤销几项，使只有其中应该有权的人才有权。例如，假设有一个 EMPLOYEES 表，而且想要每个人都对其有 SELECT 权限。但只有部门经理（ROBERT、RICHARD、LINDA 和 JULIE）应能更新工资列，只有人力资源（HR）部门的工作人员（LINDA、DOREEN 和 FRED）可以更新个人信息、删除行（删除离开的雇员）或是插入新雇员信息。最后，应该没人能够改变 EMP_ID 列的雇员 ID。

为了建立对 EMPLOYEES 表的安全性，可执行以下一系列 GRANT 语句：

```
GRANT INSERT, DELETE, SELECT ON employees
TO linda, doREEN, fred
GRANT UPDATE ON employees
  (f_name, l_name, address, city, SSAN, state, zip_code,
  phone1, phone2, email_address, emergency_contact,
  sheriff_card, bond_amount, bond_number)
TO linda, doREEN, fred
GRANT UPDATE ON employees
  (quota, bonus_rate, weekly_salary)
TO robert, richard, linda, julie
GRANT SELECT ON employees
TO lori, samantha, helen, william, james, joyce, nick,
donna, karen, amber, vivian, george
```

或者，用较少的键入，将 GRANT 和 REVOKE 语句结合起来得到相同的结果，如下所示：

```
GRANT SELECT ON employees TO public
GRANT INSERT, DELETE, UPDATE ON employees
TO linda, doREEN, fred
REVOKE UPDATE ON employees
  (emp_ID, quota, bonus_rate, weekly_salary)
FROM doREEN, linda, fred
GRANT UPDATE ON employees
  (quota, bonus_rate, weekly_salary)
TO robert, richard, linda, julie
```

请注意，使用 PUBLIC 角色向所有可访问 DBMS 的用户授予 SELECT 权限可消除在 GRANT SELECT 语句键入每个用户名的要求。同时，向 HR 工作人员授予对 EMPLOYEES 表的所有列的 UPDATE 权限，然后再使用 REVOKE 语句撤销对几个不应更新列的 UPDATE 权限，就不必在对 HR 用户的 GRANT UPDATE 语句中键入 EMPLOYEES 表中全部列名清单。

技巧 260 理解并发事务处理问题和隔离级别

当多位用户同时访问并更新数据库时，DBMS 必须保证在事务处理过程中一位用户执行的任务不会为另一用户的事务处理内的语句提供不一致的数据库值。理想地说，每位用户应该能够处理数据库而不用管其他用户的并发行动。

一般来说，如果 DBMS 不能适当地处理并发的事务处理过程的话，可出现 4 种问题：

- 丢失更新。假设 Rob 和 Lori 同时到自动取款机（ATM）从单一共有支票账户提取\$150

现款，此账户现在的余额是\$200。Rob 通过键入其账号并请求提取\$150 而开始其事务处理。ATM 用账户的当前余额响应其请求，并要求 Rob 确认提款。同时，Lori 开始其事务处理也给出余额为\$200，也被要求确认其\$150 的提款请求。如果其后每人都确认提款，Rob 的 ATM 分配\$150 并更新了账户的余额为\$50。同时，Lori 的 ATM 也分配了\$150 并更新账户余额为\$50。根据 DBMS 先完成了哪个账户的事务处理，另一人对账户余额的事务处理的效果就丢失了。这样 Rob 和 Lori 都很高兴，因为每人都提走了\$150（总共提走了\$300），而账户的余额还有\$50，而不是透支了\$100。

- 未提交的数据/作废读取。假如 Rob 和 Lori 在不同的商店内为互赠礼物而购物。他们都发现了“好”的礼物并将拿到收款台并试图用同一账户的信用卡结账，信用卡上的余额为\$750。当收款员在机器上刷卡并输入购物价值\$500 时，Lori 开始其交款的事务处理。Lori 一边等待收款员完成其购物的事务处理，一边想着这次购物过程。此时 Rob 向另一收款员交出了同一账户的付款卡，该收款员刷卡并输入\$400 的购货价值。但是，由于 Lori 已经将该账户上的可用余额减少了\$500（还仅余\$250 可用），Rob 的事务处理被拒绝。过了一会，Lori 改变了主意让收款员取消了其收费。这样一来，Rob 认为信用卡还有不到\$400 的可用款项，虽然实际上仍然有\$750 可用，因为其事务处理“看到了”从 Lori 处的未提交的更新。
- 不一致的数据/不可重复性读取。Rob 决定他确实想要他所选的那件周年礼物，因而他到 ATM 处从其储蓄中向此共用支票账户转款。他先请求查阅储蓄账记的余额，显示为\$2,000。Rob 再请求查阅支票账户的余额，显示还有\$50。而同时，Lori 去银行也从储蓄中提款以便支付\$250 的电子账单。在银行柜员完成 Lori 的提款事务处理时，Rob 继续其从储蓄存款中向支票账户转款\$750 的事务处理，正在此时，ATM 再次显示其储蓄账户的余额，此时的读数为\$1,750 而不是\$2,000。这样一来，显示在储蓄账户上的余额总是正确地反映了实际情况。但是，“有时”在 Rob 的存款转支票的转款事务处理使得 ATM 上的程序向其第一次显示一种余额，下次又显示另一种余额。换句话说，对 BALANCE 列的读取是不可重复的，因为在 Rob 的事务处理过程中不同时间显示不同的结果。
- 幻影插入/删除。现在 Rob 被储蓄和支票账户的余额弄糊涂了，ATM 显示\$1,750，这也不成问题，Rob 接着请求查阅支票账户的余额，其中仍为\$50。然后再继续转款事务处理，输入了\$750 作为从存款向支票的转款数量。同时，Lori 向支票账户中存了\$1,500。当柜员完成其存款事务处理的时间恰好在 Rob 的转款之前，正在请求查阅两个账户的余额，Rob 的事务处理就陷入了“幻影插入”（phantom insert）问题。支票账户的余额在事务处理开始时是\$50，现在加上转来的\$750，实际应为\$800。但是，显示的却是\$2,300，因为在 Rob 的事务处理时，在第二次查询支票账户余额时不知从什么地方跑出\$1,500 来。（如果，例如，Lori 使用支票存款正好在 Rob 开始其事务处理之前，然后在 Rob 转款期间提出现金，则会出现幻影删除。）

隔离级别是高级的 DBMS 锁定技术，以便在多用户环境中保持每个用户都在一人专用数据库的假象，而同时仍然让尽量多的用户同时访问数据库中的数据。虽然通过锁定所使用的表集，事务处理可防止这 4 种并发事务处理问题，这样做将使 DBMS 看起来有点迟钝，用户必须等待多个事务处理完成才能获得系统的响应。读者将在技巧 261~技巧 269 中学习有关隔离

级别的问题。现在，要理解的重要事情是，隔离级别给人们一种方式，以便告诉 DBMS，一个程序（或 SQL 语句的批处理）将不会重新提取数据，这就允许 DBMS 在事务处理结束之前释放锁定。

技巧 261 理解 READ UNCOMMITTED 和作废读取

READ UNCOMMITTED（未提交的读取）是 4 种事务处理隔离级别中最弱的一级。事实上，READ UNCOMMITTED 与没有锁定数据库完全相同。如果提交如下语句：

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

DBMS 将把会话环境设置得使事务处理不对所处理的数据加以任何共享（或专用）锁定。另外，事务处理中的该语句将不遵守其他人所掌握的锁定。因而，如果另一用户提交了改变数据库的事务处理，则查询在提交这些变化之前将看到这些变化。事实上，在 READ UNCOMMITTED 隔离级别上执行的事务处理，所有 4 种并发事务处理遇到的问题（读者已在技巧 260“理解并发事务处理问题和隔离级别”中学习过有关内容）都可能发生。

执行“作废读取”（dirty read）（即读取了更新，但却是还未提交的数据）是有问题的，因为用户更新数据库时可能会撤消未提交的事务处理并取消任何变化。结果，在 READ UNCOMMITTED 隔离级别下工作，提取并正要为事务处理所使用的数据值可能与实际存在于数据库表中的数据不匹配。

将隔离级别设置为 READ UNCOMMITTED 的主要好处是，在此环境下所处理的语句可避免并发控制所涉及的开销。DBMS 不必实行锁定（这可减少数据库管理开销），语句不必等待其他事务处理的完成（这使得提取信息更快），其他用户也不必等待自己的事务处理的完成（这使得整个系统看起来响应得更快）。遗憾的是，性能上的收益是有代价的——如果作废（未提交）的数据被撤消（取消）而且没有完成（未永久地写入数据库）时，在 READ UNCOMMITTED 环境下的事务处理结果是基于不准确的输入值的。

对于有些应用，可使用 READ UNCOMMITTED 隔离级别。如统计上不会受到 DBMS 处理事务处理的平均变化的影响的报告。例如，如果想要了解为定单发货时的平均、最小或最大延迟，或者，如果想要了解在过去的 3 个月中雇员销售的平均量或顾客购货的平均量等。

注意：大多数 DBMS 产品（如 MS-SQL Server）对连接到数据库的会话的整个时间设置隔离级别。由此，如果有一个应用程序将隔离级别设置为 READ UNCOMMITTED，在退出或是执行一段要求较高的隔离级别（更多的保护）代码段之前，请确保该程序将隔离级别设置回原来的默认级别（READ COMMITTED）。

技巧 262 理解 READ COMMITTED 和不可重复读取

READ COMMITTED（提交读取）隔离级别通过隐藏未提交的变化解决了作废读取的并发事务处理问题。这样一来，如果应用程序执行以下 SET 语句：

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
```

在会话期执行的任何语句都将只看到（处理）提交给（永久地写入）数据库的数据值。因而，如果 SUE 正在为 10 个铁锤的定单发货，而当前存货目录中有 20 把铁锤，她的程序执行以下 UPDATE 语句：

```
UPDATE inventory SET qty = qty - 10
```

```
WHERE description = 'hammer'
```

她的以下查询将返回 QTY 为 20（而不是 10）：

```
SELECT qty FROM inventory WHERE description = 'hammer'
```

直到那时 SUE 的应用程序结束或是执行了 COMMIT 语句，DBMS 将 SUE 的 UPDATE 语句提交到 INVENTORY 表时，QTY 列的值才会为 10。

READ COMMITTED 隔离级别有趣的一面是，事务处理可能会结束处理在数据库中找不到的数据。例如，假设工资发放部门的 MARY 正在执行周期结束的关闭步骤，其中执行以下 DELETE 语句来删除在正在关闭的发工资期间插入到 TIMECARDS 表中的行：

```
DELETE FROM timecards
```

```
WHERE card_date BETWEEN @period_start and @period_end
```

如果运行查询并报告在 @PERIOD_START 和 @PERIOD_END 之间所包括的带 CARD_DATE 列的 TIMECARDS 表行，应用程序将继续“看到”这些行，即使 SUE 的删除语句已经将其从 TIMECARDS 表中删除。

现在，在 SUE 应用程序执行 COMMIT 语句并结束 DELETE 语句的操作时，删除的 TIMECARDS 表行才对应用程序“消失”。这样，READ COMMITTED 隔离级别不能防止幻影插入或幻影删除等并发事务处理问题。

另外，虽然 READ COMMITTED 消除了作废读取，从而防止了不一致的数据问题，但它并不能防止不可重复的读取。例如，假设 MARY 和 SUE 每人都正在从顾客处获取定单。MARY 查询 INVENTORY 表后发现，她有 25 本 1001 FrontPage 2000 Tips 并让其顾客了解。同时，SUE 完成了其对 20 本该书的定单，而且 DBMS 将其工作提交给 INVENTORY 表。当 MARY 的顾客说：“我要 25 本”时，她发现不能处理这个定单，因为第二查询显示可卖的书只有 5 本了。因此，同一查询在一次销售事务处理期间得到了不同的结果，前次读取（查询）是不可重复的。

技巧 263 理解 REPEATABLE READ 和幻影插入

REPEATABLE READ（可重复读取）在 4 种隔离级别的层次中处于 READ COMMITTED 的一步之上，既可用于防止作废读取问题也可防止（顾名思义）不可重复读取问题。不是简单地忽略所有未提交的更新和删除，而是 DBMS 实施锁定，防止其他用户改变由 REPEATABLE READ 隔离级别下的事务处理所读取的数据。另外，如果运行在 REPEATABLE READ 隔离级别上的事务处理不能锁定需要查询的部分数据库（由于另一用户的事务处理已经将其锁定），DBMS 就得等待锁定释放才能执行事务处理。

这样一来，前面技巧中的示例，其中 MARY 和 SUE 都在接手相等的 25 本 1001 FrontPage 2000 Tips 书的定单，可重复读取问题就出现了，因为 MARY 和 SUE 两人都同时访问 INVENTORY 表的同一行数据。如果定单输入程序是在 REPEATABLE READ 隔离级别下执行的（而不是在 READ COMMITTED 隔离级别下执行），那么一次两人中只有一个人能够使用 INVENTORY 表的特定部分。

因而，如果定单输入程序运行在 REPEATABLE READ 隔离级别下，MARY 对书的本数的请求锁定了部分 INVENTORY 表，使其不能修改，而且也不能回答其他在同样的或更高隔离级别下的查询请求。因而，当 SUE 稍后开始对同一种书籍的事务处理时（这将涉及读取与 MARY 锁定的 INVENTORY 表的同一部分），DBMS 拒绝 SUE 的事务处理的锁定请求。结果，

SUE 的事务处理必须等待，直到 MARY 完成其工作并释放对 INVENTORY 表的部分锁定，以便 SUE 的事务处理能够读取。

由于 REPEATABLE READ 隔离级别只防止其他用户修改开放的事务处理已经读取的数据，运行于 REPEATABLE READ 隔离级别上的事务处理仍然会发生幻影插入（以及删除）问题。例如，假设执行如下的批处理语句：

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
SELECT SUM(order_total) FROM orders WHERE cust_ID = 101
SELECT * FROM orders WHERE cust_ID = 101
```

如果在由 SELECT 语句（批处理中的第 3 条语句）返回的行的 ORDER_TOTAL 列上增加了值的话，由总计函数 SUM(ORDER_TOTAL)返回的值与获得的总数不同也是可能的。如果在 DBMS 正在计算 ORDER_TOTAL 列的总和期间，用户向 ORDERS 表中插入了顾客 CUST_ID 101 的附加定单而 SELECT 语句显示的正是 ORDERS 表的 CUST_ID 101 行，这个错误或不正常就会出现。

技巧 264 理解 MS-SQL Server 的锁定扩大

众所周知，保持行级锁定比保持页级锁定需要更多的开销，而页级锁定又比表级锁定需要更多的开销。在默认情况下，MS-SQL Server 将在行级锁定数据。因而由系统执行的任何查询都至少锁定表中的一行。当系统执行事务处理中的语句时，DBMS 在每 8KB 页数据上跟踪它所保持的锁定数目。在某一时刻，根据每页锁定行的百分数，DBMS 将扩大锁定级别，从而开始使用页级锁定而不是行级锁定。

例如，如果事务处理执行下面的 UPDATE 语句：

```
UPDATE employees SET hourly_rate = hourly_rate * 1.2
WHERE office = 1
```

若 EMPLOYEES 表中每 8KB 页面上只有几行，同时 OFFICE 列值为 1，DBMS 将锁定单独的 EMPLOYEES 表行。另一方面，若事务处理执行以下查询：

```
SELECT * FROM orders
WHERE order_date BETWEEN '01/01/2000' AND '12/31/2000'
```

其中要在页面上读取（并锁定）大量表行，则 DBMS 将一次锁定 ORDERS 表的一个 8KB 的页。另外，如果事务处理使用如下语句超过了表级扩大阈值，最终将锁定表中的每行和每页，DBMS 将对 CUSTOMERS 表使用表级锁定：

```
SELECT * FROM customers
```

所以锁定扩大（Lock escalation）是将大量更细的行级锁定转换为较少但较粗的页级锁定，如果需要的话，还会转换为表级锁定，这一切都是为了减少系统开销。MS-SQL Server 动态地设置锁定的扩大阈值，这样一来，用户不需要对此加以配置。例如，当事务处理需要一个表中的行时，MS-SQL Server 锁定事务处理中语句使用的行并将对页级锁定意向放在包括锁定的行的页上。如果后来事务处理超过了这个页面阈值，DBMS 就试图将较高级别的意向锁定改变为实际的锁定。在获得了（较少的）页级锁定之后，DBMS 去掉（较多的）行级锁定，因而减少了系统锁定的开销（当从页级锁定扩大到表级锁定时，DBMS 遵照同样的步骤）。

这样一来，当 SQL 语句引用散布于大型表中的较小数目的行时，MS-SQL Server 最大化并发访问数目，直到数据变为行级锁定为止。但是，如果语句引用了表中的大多数或所有的行，

系统通过切换到表级锁定从而最大化并发访问数。由于 MS-SQL Server 动态地调节其锁定细度，单条语句可能会对一个表使用行级锁定，而对另一表使用页级锁定，对第 3 个表使用表级锁定。

技巧 265 理解死锁以及 DBMS 如何解决死锁

死锁是两个用户（或会话）都锁定了不同的对象，而且每个用户的事务处理都正在等待对另一用户锁定的对象的锁定的情况。图 14.1 说明了死锁条件，其中事务处理 A 锁定了 ORDERS 表，而且需要锁定 CUSTOMERS 表以便处理。同时，事务处理 B 锁定了 CUSTOMERS 表，正在等待事务处理 A 完成并释放其对 ORDERS 表的锁定。

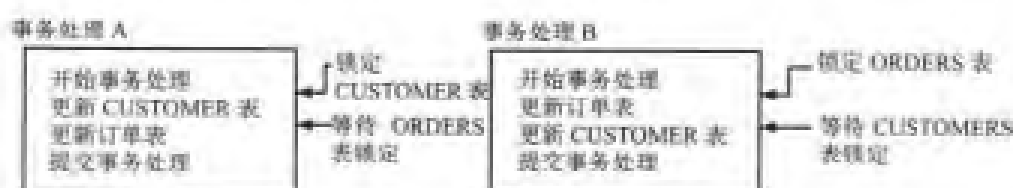


图 14.1 事务处理 A 和事务处理 B 都被死锁，正在等待另一事务处理在处理之前释放表

如果没有 DBMS 的干预，每个会话都将永远等待下去，等待另一事务处理提交或取消（COMMIT 或 ROLLBACK）其事务处理并解除对部分数据库文件的锁定，以便会话可继续下去。虽然图 14.1 描述了简单的两个事务处理的死锁，但在实际当中，死锁的情况往往更为复杂。3 个或更多的会话陷入锁定的死循环中，每个都在等待被一个或多个其他会话锁定的数据。

为了处理死锁问题，DBMS 将选择会话之一作为“死锁受害者”将其中止，取消由此事务处理所做的改变并释放其锁定，从而其他会话中的可以继续下去。

每种 DBMS 有其自己的消除死锁以及选择“死锁受害者”的方法。例如，MS-SQL Server 周期性地扫描并标注等待锁定请求的会话。如果在下一周期扫描中，标注的会话仍然等待锁定，则 DBMS 开始递归性的死锁搜索（毕竟，会话可能刚刚被阻塞，正在等待一个大型查询或其他长时间运行的 SQL 语句的完成。）如果递归性的死锁搜索找到了一个锁定请求循环链，MS-SQL Server 选择最小量的事务处理来取消其所做工作并将其终止。

MS-SQL Server 不仅取消死锁的受害者的事务处理，而且 DBMS 还发送 1205 号错误消息通知用户程序事务处理不能进行。然后服务器取消终止的事务处理的锁定请求并允许未终止的事务处理继续进行下去。

由于 DBMS 处理死锁的方式不同，任何 SQL 语句都是潜在的死锁受害者。因此，批处理执行 SQL 语句的应用程序必须检查而且要能够处理使用多个表的任何事务处理中的死锁受害者错误代码。如果死锁受害者错误代码是在交互式会话中返回的，用户可以简单地重新键入并重新提交那些 SQL 语句。在程式化 SQL 中，接受死锁受害者错误代码的应用程序既可警告用户并终止事务处理，也可自动地重新开始事务处理。

技巧 266 理解 SERIALIZABLE 隔离级别

可串行化（Serializable）是一种 DBMS 概念，其意义是，每个数据库用户可访问数据库，就好像没有其他用户同时访问同样的数据一样。简单地说，串行性要求用户的应用程序在事务处理中将能看到完整而一致的数据视图，而且不管其他用户未提交的还是提交的操作都不会影

响事务处理语句所看到的数据值。或者换一种说法, SERIALIZABLE 指的是, 如果两个事务处理(事务处理 A 和事务处理 B)正在并发执行, DBMS 将保证不管是事务处理 A 先执行还是事务处理 B 先执行, 结果都是一样的。

正如表 14.1 所示, 如果执行以下 SET 语句。

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

表 14.1 SERIALIZABLE 隔离级别和并发事务处理更新问题

隔离级别	丢失的更新	未提交的数据/ 作废读取	不一致的数据/ 不可重复的读取	幻影插入
可串行化	已防止	已防止	已防止	已防止

DBMS 将保证会话期中执行的事务处理不会出现四种并发事务处理问题(读者已在技巧 260 “理解并发事务处理问题和隔离级别”中学习了有关内容)。

简单地说, SERIALIZABLE 完全隔离了事务处理, 使之不受 DBMS 代表其他用户并发执行的 SQL 语句的影响。

当会话处于 SERIALIZABLE 隔离级别时, DBMS 在整个事务处理存在期间将保持每个事务处理对其使用的更新的部分数据库表的专门锁定。在 SERIALIZABLE 隔离级别上开始执行事务处理之后, DBMS 将不允许任何其他用户对所读取的数据或由事务处理中的语句所修改的数据做任何改变。另外, DBMS 还将防止用户在事务处理语句中所执行的任何 WHERE 子句的结果集中插入新的数据行, 也不能执行删除任何这样行的 DELETE 语句。

技巧 267 理解 REPEATABLE READ 隔离级别

REPEATABLE READ (可重复读取) 隔离级别是第二高的隔离级别(只比 SERIALIZABLE 低一级)。在 REPEATABLE READ 隔离级别上, DBMS 防止第二个事务处理修改由第一个事务处理所读取的数据, 直到第一个事务处理结束为止(并释放对所读取数据的共享锁定)。但是, 另一用户可向第一个事务处理查询的表中插入附加行。因此, 一个如下的查询:

```
SELECT * FROM orders WHERE cust_ID = 2002
```

在事务处理 A 开始时执行与在同一事务处理中稍后执行, 可能会有不同的结果——即使事务处理本身并不对 ORDERS 做任何改变。如果另一用户执行一条向 ORDERS 表添加 CUST_ID 列值为 2002 的附加行时, 就会出现由于幻影插入问题引起的不一致查询结果。

大多数应用程序在单一事务处理期间并不需要重复多行查询的能力。因而, 只要可能就让会话保持在 READ COMMITTED 隔离级别(MS-SQL Server 的默认隔离级别)上, 因为在 REPEATABLE READ 隔离级别上, 对其他用户明确地数据锁定会降低数据库的并发性。在大多数情况下, 应用程序绝不会“看到”新插入的行(因为在同一事务处理中并未在多处执行同一查询), 而且 DBMS 将保持自身的一致性, 只要在事务处理期间的某时所读取的数据值在整个事务处理过程中保持不会受到其他用户的改变。

正如表 14.2 所示, REPEATABLE READ 隔离级别防止 4 种并发(多用户)更新问题中的 3 种。

在 REPEATABLE READ 隔离级别上执行比在 SERIALIZATION 隔离级别上执行的主要好处是改善了系统的总体性能。如果事务处理 A 正在 REPEATABLE READ 隔离级别上执行而其

他用户提交了 INSERT 语句向事务处理使用的表中添加行，DBMS 将在继续处理事务处理 A 中的语句的情况下添加行。另一方面，如果事务处理 A 运行在 SERIALIZABLE 隔离级别，任何向事务处理 A 中已经执行的任何语句中的 WHERE 子句返回的结果表中添加行的 INSERT 语句都必须等待，直到 DBMS 完成事务处理 A 中的处理为止。

表 14.2 SERIALIZABLE 和 REPEATABLE READ 隔离级别以及并发事务处理更新问题

隔离级别	丢失的更新	未提交数据/ 作废读取	不一致数据/ 不可重复读取	幻影插入
可串行化	防止	防止	防止	防止
可重复读取	防止	防止	防止	可能

技巧 268 理解 READ COMMITTED 隔离级别

虽然 READ COMMITTED（提交的读取）隔离级别只防止 4 种并发事务处理问题中的两种（如表 14.3 所示），在隔离级别分支上的第 3 高的隔离级别通过当数据一开始读取就释放共享锁定来改善数据库的并发性。

表 14.3 SERIALIZABLE、REPEATABLE READ 和 READ COMMITTED

隔离级别和并发事务处理更新问题

隔离级别	丢失的更新	未提交数据/ 作废读取	不一致数据/ 不可重复读取	幻影插入
可串行化	防止	防止	防止	防止
可重复读取	防止	防止	防止	可能
提交的读取	防止	防止	可能	可能

DBMS 不允许事务处理中在 READ COMMITTED 隔离级别上执行的语句看到由其他并发执行的事务处理所做的未提交的更新，从而防止出现未提交数据/作废读取问题。但是，不可重复读取和幻影插入问题可能出现，因为由其他事务处理所做的提交的变化在事务处理过程中可能会变为可见的。结果，重复如下查询：

```
SELECT cust_ID, total_orders FROM customers
WHERE cust_ID = 3003
```

当代表其他用户并发执行的事务处理改变了 CUSTOMERS 表中 CUST_ID 列值为 3003 的值并执行 COMMIT 语句将改变永久地写入表中时，事务处理每次返回不同的 TOTAL_ORDERS 值。类似地，一个不止一次执行如下 SELECT 语句的事务处理：

```
SELECT COUNT(*) FROM orders WHERE cust_ID = 4004
```

可能会发现，总计函数 COUNT(*) 可能每次会返回不同的值，因为其他事务处理，如 COMMIT、INSERT 和（或）DELETE 语句添加或删除 ORDERS 表中 CUST_ID 列值为 4004 的行。

除了通过隐藏未提交的变化而避免作废读取之外，READ COMMITTED 隔离级别可防止丢失的更新。例如，如果事务处理 A 中的试图更新已经由事务处理 B 改变的行，DBMS 将自动地取消由事务处理 A 所做一切变化。因此，如果编写一个执行几个 UPDATE 语句的应用程序，一定要在每个语句之后执行 COMMIT 语句。否则，如果程序偶尔试图更新已经由另一并发执行的

事务处理所更新的行时，DBMS 将取消应用程序开始以来所做的改变。

技巧 269 理解 READ UNCOMMITTED 隔离级别

READ UNCOMMITTED（未提交读取）是 SQL-92 标准定义的最低一级的隔离级别并且（如表 14.4 所示）只能防止丢失更新问题。事实上，在某些 DBMS 产品上（如 MS-SQL Server），READ UNCOMMITTED 隔离级别甚至不能防止丢失更新问题。因而，一定要检查系统文档，看一看 READ UNCOMMITTED 是否像 SQL-92 规范所指明的，意味着“READ ONLY”（只读），因而不允许在 READ UNCOMMITTED 隔离级别下执行的事务处理的更新操作。

表 14.4 SERIALIZABLE、REPEATABLE READ、READ COMMITTED 和 READ UNCOMMITTED
隔离级别和并发事务处理更新问题

隔离级别	丢失的更新	未提交数据/ 作废读取	不一致数据/ 不可重复读取	幻影插入
可串行化	防止	防止	防止	防止
可重复读取	防止	防止	防止	可能
提交的读取	防止	防止	可能	可能
未提交的读取	防止	可能	可能	可能

如果在具体的 DBMS 上 READ UNCOMMITTED 意味着 READ ONLY（只读）模式，那么在 READ UNCOMMITTED 隔离级别下，DBMS 将防止出现丢失更新问题。因为事务处理在 READ UNCOMMITTED 隔离级别下不能执行任何类型的更新。但是，如果 DBMS 产品像 MS-SQL Server 一样，允许在 READ UNCOMMITTED 隔离级别下更新，如果两个并发运行的事务处理执行改变同一表行和列中的数据值的 UPDATE 语句，则会出现丢失的更新问题。由于不管事务处理请求或保持的共享还是专门锁定其修改的数据，都不能防止由其他事务处理产生的并发变化，由一条 UPDATE 语句改变的数据值可立即被第二条 UPDATE 语句加以改变，甚至在 DBMS 完成其第一条 UPDATE 语句的执行之前也是如此。

在 READ UNCOMMITTED 隔离级别下执行 SELECT 语句的主要好处是，这样做允许最高程度的数据库并发性，因为在 READ UNCOMMITTED 模式下执行的查询不仅不锁定所读取的数据，而且还忽略任何由其他事务处理所保持的已有的共享或专有锁定。但是，为了绕过系统的并发处理保护，如下的 SELECT 语句，在查询对 INVENTORY 表中那些更新的但还未提交的数据值执行作废读取时，可能会报告不正确的结果：

```
SELECT item_number, qty_on_hand inventory
```

例如，如果 DBMS 执行一条未提交的 ROLLBACK 语句，接着中止与 SELECT 语句并发执行的存货目录的更新或在线定单输入事务处理，则查询的结果表将包括从未在数据库中实际存在过的值。

因而，仅对那些结果不必百分之百准确的查询，如在一段时间以来所处理的定单的 COUNT(*)，或是对于那些寻找趋势或不关心准确的特定行或行集中的值时，才可使用 READ UNCOMMITTED 隔离级别。另外，为了避免丢失的更新问题，不要在 READ UNCOMMITTED 隔离级别下执行 UPDATE 语句，即使具体的 DBMS 产品允许这样做。

技巧 270 使用 MS-SQL Server Enterprise Manager 显示阻塞和被阻塞的会话

正如读者在技巧 265 “理解死锁以及 DBMS 如何解决死锁”中所学习的，MS-SQL Server 通过终止一种死锁的事务处理，从而释放锁定使其他事务处理可继续运行，自动地解决死锁问题。但是，一个会话可阻塞其他应用程序，使其在一段时间内不能读取或更新一个或多个表，而不会形成 DBMS 将要清除的死锁。例如，假设 FRANK 启动了 MS-SQL Server Query Analyzer 会话并执行以下语句批处理：

```
BEGIN TRANSACTION
SELECT * FROM auto_inventory
UPDATE auto_inventory SET MODEL = 'Model 2'
WHERE year = 1999 AND make = 'Camero'
```

MS-SQL Server 将给予 FRANK 的事务处理对由其 UPDATE 语句修改的 AUTO_INVENTORY 表行以专有锁定。另外，FRANK 的处理将保留这些锁定，直到结束会话或执行了 ROLLBACK 或 COMMIT 语句而结束事务处理为止。

如果另一过程试图查询或更新 AUTO_INVENTORY 表，事务处理将“挂起”等待 FRANK 的会话释放其专有的锁定。而且，DBMS 既不会结束 FRANK 的事务处理，也不会结束其他查询（或 UPDATE）事务处理，因而两个过程没有死锁。实际情况是，FRANK 的过程只是正在阻塞其他对共享或专有的对 AUTO_INVENTORY 表行的锁定可使用 MS-SQL Server Enterprise Manager 来显示阻塞和被阻塞的会话，为达到目的，请执行以下步骤：

1. 为了启动 Enterprise Manager，可单击 Start 按钮，将鼠标指针移动到 Start 菜单的 Programs 上并选择 Microsoft SQL Server 7.0，然后单击 Enterprise Manager。
2. 为了显示 SQL 服务器的列表，可单击 SQL Server Group 左边的加号（+）。
3. 找出管理着想要显示显示会话的 SQL 服务器，单击 SQL Server 图标左边的加号（+）。例如，如果想要显示运行在名为 NVBizNet2 的 SQL Server 上的会话，可单击 NVBizNet2 图标左边的加号（+）。Enterprise Manager 将显示包括在 MS-SQL Server 上可用的数据库和服务的文件夹。
4. 单击 Management 文件夹左边的加号（+）。Enterprise Manager 将显示各种数据库管理活动和状态报告的图标。
5. 单击 Current Activity 左边的加号（+）。Enterprise Manager 将显示可用于显示进程信息和锁定的图标（通过进程 ID 和对象 ID）。
6. 双击 Locks/Process ID 左边的图标。Enterprise manager 接着显示当前运行的系统进程 ID（SPID），如图 14.2 所示。

请注意，Enterprise Manager 窗口的右窗格显示哪个会话正在阻塞另一会话中的事务处理过程。在本例中，Enterprise Manager 显示 SA 的会话（SPID 7）已被 FRANK 的会话（SPID 9）阻塞。为了显示由一个会话所执行的批处理中的最后一个语句，可右击想要看到的最后批处理语句的会话图标，然后从 Enterprise Manager 的弹出菜单中选择 Properties。Enterprise manager 接着显示 Process Details（进程详情）对话框，如图 14.3 所示。

请检查由阻塞会话所执行的最后一条语句，决定用户是否正在执行一个需要花很长时间才能完成的进程，或者（正如本例中的情况一样）用户是否不经意地没有执行 COMMIT 语句来关闭打开的事务处理并释放其对数据库对象的锁定。当找出为什么会话会阻塞另一会话之后，

可耐心地等待阻塞进程的完成、与阻塞会话的拥有者联系并请该用户完成或中止其打开的会话，或者直接结束该阻塞会话（读者将在技巧 271“使用 MS-SQL Server Enterprise Manager ‘杀死’对数据库对象保持锁定的进程”中学习有关知识）。

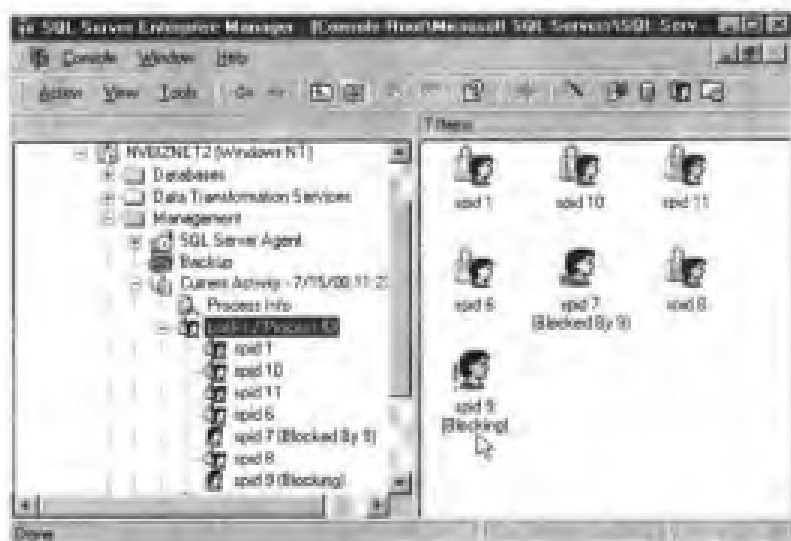


图 14.2 MS-SQL Server Enterprise Manager Locks/Process ID 窗口

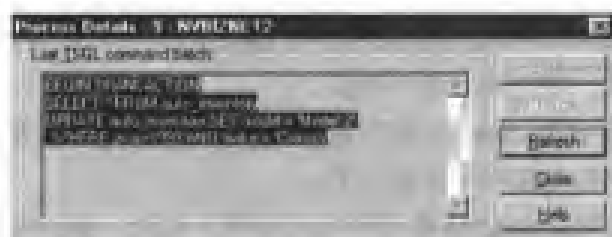


图 14.3 MS-SQL Server Process Details 对话框

为了退出 Process Details 对话框，可单击关闭按钮。

技巧 271 使用 MS-SQL Server Enterprise Manager “杀死”对数据库对象保持锁定的进程

在技巧 270“使用 MS-SQL Server Enterprise Manager 显示阻塞和被阻塞的会话”中，读者已经学习了如何使用 MS-SQL Server Enterprise Manager 来确定正在阻塞其他会话的会话的进程 ID（SPID）。如果认为一个事务处理在合理的时间不会完成其工作，可利用 Enterprise Manager 将其中止（例如，假设 FRANK 启动了对 AUTO_INVENTORY 表更新的会话，然后在执行 COMMIT 语句关闭事务处理使变化成为永久的并释放该事务处理所保持的锁定之前就去吃午饭了）。

为了使用 MS-SQL Server Enterprise Manager 来结束会话中打开的事务处理，请执行以下步骤：

1. 遵照技巧 270 中的 6 个步骤显示运行在 MSSQL Server 上的会话的清单，以便决定哪个会话阻塞了其他会话。

2. 单击 Process Info（进程信息）左边的图标，显示有关运行在 MS-SQL Server 上的第 1 步中所选进程的信息（如图 14.4 所示，在 MS-SQL Server Enterprise Manager 的左窗格中，Process

Info 图标紧列在 Locks/Process ID 图标之上)。

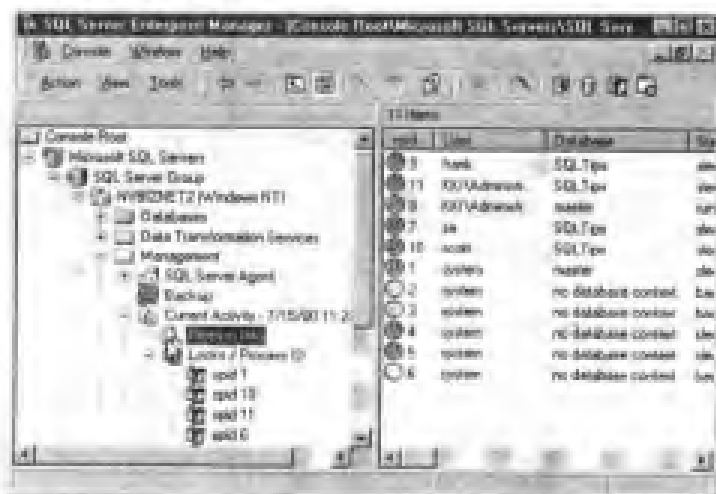


图 14.4 MS-SQL Server Enterprise Manager Process Info 窗口

3. 右击想要中止(“杀死”)的进程的 SPID 左边的图标。对本例来说, FRANK 的事务处理(SPID 9)正在阻塞 SA 的查询(SPID 7), 可在 Enterprise Manager 的右窗格内右键单击 SPID 列中值为 9 的地球图标。

4. 从弹出菜单中选择 Kill Process (“杀死”进程)选项。Enterprise Manager 将显示 Kill Process 消息框提示用户确认“杀死”在步骤 3 中选定的 SPID 中运行的进程的意向。

5. 为了“杀死”进程, 可单击 Yes 按钮。

完成步骤 5 之后, MS-SQL Server 将中止运行在会话中 SPID 为在步骤 3 中所选定的值的会话并取消该会话中已经完成的未提交的工作。

注意: 除了 Enterprise Manager 的图形界面(GUI)之外, MS-SQL Server 还提供了可在命令行上执行的存储的例程和 SQL 语句, 以便找出阻塞进程的 SPID 并将其中止。如果执行存储过程 sp_lock, MS-SQL Server 将显示已经阻塞和正在请求锁定(也就是被阻塞的 SPID)的 SPID 的清单。在确定了想要中止的运行事务处理的会话的 SPID 之后, 可执行以下语句将该事务处理中止:

```
KILL <SPID>
```

其中<SPID>是想要“杀死”的进程的 SPID 号。

技巧 272 理解 MS-SQL Server 与 Oracle 上的锁定和事务处理隔离

通过对技巧 260~技巧 269 中所描述的复杂锁定机制和隔离级别的复习, 可以看到 DBMS 产品作出相当的努力来处理多用户并发的查询与更新, 同时将用户与其他用户的操作相互隔离。事实上, DBMS 的主要功能之一是确保多个用户可查询和更新数据库, 而没有接受到正在进行的事务处理的不一致的信息, 也没有不经意地互相改写更新。但是, 虽然所有 DBMS 产品的目的都是一样的, 但每种产品所使用的互相保护的方法却可以有很大的不同。

例如, Oracle 和 MS-SQL Server 就对数据库锁定和隔离策略采取了很不一样的方法。在 Oracle 上, 读取数据的语句在读取数据行之前, 既不能得到锁定也不等待锁定释放。因此, 在技巧 270“使用 MS-SQL Server Enterprise Manager 显示阻塞和被阻塞的会话”中经常被阻塞的正如本技巧所述的进程不会出现。在 Oracle 的 DBMS 上, FRANK 的事务处理中的 UPDATE

语句仍然会获得对 AUTO_INVENTORY 表中的一行（或许一页）的专有锁定。但是，系统管理员（SA）的会话中的查询却不会“挂起”。在 Oracle 环境中，SA 的查询会读取包括 FRANK 的未提交的变化之前的数据值的数据副本，而不是等待当 FRANK 的会话提交其工作时对当前大多数数据值的 READ COMMITTED 访问。

因而，在 Oracle 的 DBMS 上，当 SELECT 语句请求已经变化但另一事务处理还未提交的数据时，系统为查询提供在查询开始时已经向数据库提交的最新数据值集。执行 INSERT、DELETE 或 UPDATE 语句的会话获得对所有修改的锁定并保持这些锁定，直到事务处理结束，以防止其他人改写未提交的变化。

另一方面，MS-SQL Server 使用共享锁定，以保证查询只看到提交的数据（当然，除非将事务处理的隔离级别设置为 READ UNCOMMITTED）。因此一条在 MS-SQL Server 上执行的 SELECT 语句，随着为结果表提取数据，就获得并释放共享锁定。虽然这些共享锁定对其他 SELECT 语句没有影响，DBMS 并不要求查询在读取更新了却还未提交的数据之前等待事务处理提交其未决的变化。由此，快速地释放锁定并减少修改数据的事务处理的时间长度在 MS-SQL Server 上比在 Oracle 服务器上更为重要。

要理解的重要事情是，必须检查具体的 DBMS 产品的锁定和隔离方法，以便保证编写出能够支持不断增加的并发用户数目的应用程序。当将 SQL 代码从一个 DBMS 产品移植到另一个产品时，复核代码是特别重要的。正如在本技巧中对 MS-SQL Server 和 Oracle 的锁定策略的比较中所学到的，一条批处理的 SQL 语句在 Oracle DBMS 上运行得很好，但若不加修改地运行在 MS-SQL Server 上则会引起多个锁定（因而用户不会满意）。

技巧 273 使用 SET TRANSACTION 语句设置事务处理的隔离级别

事务处理的隔离级别告诉 DBMS 想让系统将自己所做的工作与 DBMS 代表其他用户并发执行的工作之间的相互隔离的程度。每种 DBMS 产品都有其默认的隔离级别。例如，MS-SQL Server 使用 READ COMMITTED 作为默认的隔离级别。SET TRANSACTION 语句允许改变会话的隔离级别。

在技巧 260 “理解并发事务处理问题和隔离级别”中，读者了解到，并发地处理多个事务处理使数据库易出现 4 个主要问题。DBMS 以 4 种不同的隔离级别实现不同的锁定机制，可防止一个或多个这样的问题，如表 14.5 所示。

表 14.5 隔离级别和并发事务处理问题

隔离级别	丢失的更新	未提交数据/ 作废读取	不一致数据/ 不可重复读取	幻影插入
可串行化	防止	防止	防止	防止
可重复读取	防止	防止	防止	可能
提交的读取	防止	防止	可能	可能
未提交的读取	防止	可能	可能	可能

SET TRANSACTION 语句的句法如下：

```
SET TRANSACTION [{READ ONLY|READ WRITE}]
ISOLATION LEVEL (READ UNCOMMITTED|READ COMMITTED|
```

```
REPEATABLE READ|SERIALIZABLE|  
[DIAGNOSTICS SIZE <number of error messages>]
```

因而，为了告诉 DBMS 想要事务处理中的语句对只防止丢失更新问题的数据库不做任何改变，可执行如下的 SET TRANSACTION 语句：

```
SET TRANSACTION READ ONLY ISOLATION LEVEL READ UNCOMMITTED
```

类似地，为了保护更新数据库的事务处理不发生所有 4 种更新并发问题并返回至多 3 种错误信息（如果必要的话），可执行以下 SET TRANSACTION 语句：

```
SET TRANSACTION READ WRITE ISOLATION LEVEL SERIALIZABLE  
DIAGNOSTICS SIZE 3
```

注意：与许多在 SQL-92 标准中定义的其他语句一样，某些 DBMS 产品支持 SET TRANSACTION 的方式并不严格地依照 SQL-92 规范。例如，MS-SQL Server 给出的 SET TRANSACTION 语句的句法为：

```
SET TRANSACTION ISOLATION LEVEL (READ UNCOMMITTED|  
READ COMMITTED|REPEATABLE READ|SERIALIZABLE)
```

这就取消了语句将事务处理访问模式设置为 READ ONLY 或 READ WRITE 的能力，从而可控制服务器在执行事务处理中的语句时遇到错误可返回的错误信息的数目。

一定要检查系统文档，以便确定具体的 DBMS 产品使用 SET TRANSACTION 语句的规则。某些 DBMS 产品，与 SQL-92 规范一样，要求在事务处理中把 SET TRANSACTION 语句作为第一条语句来执行。除此之外，这些 DBMS 产品使用语句的子句来为语句出现的事务处理改变默认的事务处理设置。另一方面，MSSQL Server 允许在事务处理中的任何地方执行 SET TRANSACTION 语句并使用 TRANSACTION ISOLATION LEVEL 子句后面的值来设置会话中余下语句的隔离级别——而不只是在单一事务处理中执行的那些语句。

注意：与大多数其他 DBMS 产品不同，在一个用户（或应用程序）在会话中通过执行 SET TRANSACTION 语句改变了隔离级别之后，MS-SQL Server 并不将事务处理的隔离级别返回系统的默认级别（READ COMMITTED）。因而，如果对会话中余下的语句将隔离级别降低为 READ UNCOMMITTED，除非执行另一条将会话的隔离级别提高到初始默认设置（READ ONLY）或以上的 SET TRANSACTION 语句，否则 MS-SQL Server 将在 READ UNCOMMITTED 隔离级别上处理事务处理过程。

技巧 274 使用 COMMIT 语句使数据库更新成为永久的

SQL-92 规范定义了两条可用来结束一个事务处理的语句：COMMIT 和 ROLLBACK。虽然执行 ROLLBACK 语句可取消一个事务处理中所完成的工作，但提交 COMMIT 语句使自从事务处理开始以来完成的所有数据修改（所有的 UPDATE、INSERT 和 DELETE 的行为）成为永久的数据库部分。另外，提交一个事务处理之后，DBMS 释放会话所掌握的所有资源（如表、页和行的锁定）。

直到执行 COMMIT 语句之时，都可执行 ROLLBACK 语句取消在事务处理内由提交给 DBMS 的语句所采取的行动。例如，假设开始了一个 MS-SQL Server Query Analyzer 会话并执行以下语句系列：

```
BEGIN TRAN  
UPDATE employees SET salary = salary * 1.5
```

```
DELETE employees WHERE office = 1
UPDATE employees SET manager = NULL WHERE manager = 102
```

如果接着执行以下语句：

```
ROLLBACK
```

DBMS 将使 EMPLOYEES 表看起来与在执行 BEGIN TRAN 之后的第一条 UPDATE 语句语句之前的一样——其中包括由 DELETE 语句从表中删除的行。但是，如果用以下语句跟随开始的语句系列：

```
COMMIT
ROLLBACK
```

则 COMMIT 语句关闭事务处理并使更新的行删除成为永久的，接下来的 ROLLBACK 就不能取消已提交（现已关闭）的事务处理所完成的工作。

如果具体的 DBMS 产品允许嵌套事务处理，执行关闭内（嵌套的）事务处理的 COMMIT 语句并不释放资源或使变化成为永久的。只有结束最外层事务处理的 COMMIT 语句实际上使对数据库的改变不可取消。例如，假设执行了以下语句批处理：

```
BEGIN TRAN outermost_tran
UPDATE employees SET salary = salary * 1.5
BEGIN TRAN nested_tran1
DELETE employees WHERE office = 1
BEGIN TRAN nested_tran2
UPDATE employees SET manager = NULL
WHERE manager = 102
COMMIT TRAN nested_tran2 - the DBMS commits no work
COMMIT TRAN nested_tran1 - the DBMS commits no work
COMMIT outermost_tran - the DBMS commits ALL work
```

只有最后的 COMMIT 语句（在外层的事务处理中）实际上使外层和内层（嵌套）的事务处理所完成的工作成为永久的数据库部分。因此，嵌套的事务处理提供了一种方便的方法，可创建一组可取消其工作的语句（一组语句一组语句地进行），而保留对是否提交整个所完成的工作体的决定，直到提交（COMMIT）或取消（ROLLBACK）最外层的事务处理为止。

注意：交互式的 DBMS 工具，如 MS-SQL Server Query Analyzer，通常将输入和执行（按 F5 键或选择 Query 菜单中的 Execute 选项）看作是隐含的事务处理。因此，如果键入以下语句：

```
DELETE employees
```

然后告诉 Query Analyzer 执行此语句，此时就不能再键入：

```
ROLLBACK
```

或按 F5 键来撤消由 DELETE 语句所完成的工作。

Query Analyzer 执行在每个语句系列（或每条单独的语句）之后执行隐含的 COMMIT 语句。为了防止交互式的 DBMS 工具（如 MS-SQL Server Query Analyzer）在每条语句之后提交所作工作，必须执行 BEGIN TRAN（或 BEGIN TRANSACTION）语句明显地标记事务处理的开始。这样做之后，Query Analyzer 将把单独的语句和语句组看作为单个的、打开的事务处理。然后就可以执行 ROLLBACK 语句取消最近的 BEGIN TRAN 以来所完成的工作，或者执行 COMMIT 语句使工作成为永久的数据库部分（如果在对每个打开的事务处理执行 COMMIT 语句之前关闭了 Query Analyzer 会话，Query Analyzer 将执行隐含的 COMMIT 语句并关闭所有打开的事务处理并使所完成的工作成为永久的）。

技巧 275 使用 SET CONSTRAINTS 语句在提交事务处理之前延缓 DEFERRABLE 约束

在一个事务处理中系统执行了所有的语句之后，而不是每执行完事务处理中的一条语句之后，可使用 SET CONSTRAINTS 语句告诉 DBMS 检查 DEFERRABLE 约束。例如，假设有一个由以下 CREATE TABLE 语句创建的 INVENTORY 表：

```
CREATE TABLE inventory
(item_number INTEGER UNIQUE DEFERRABLE,
 item_cost    MONEY,
 description  VARCHAR(30),
 CONSTRAINT non_zero_cost
 CHECK (item_cost > 0) DEFERRABLE)
```

就指明了 ITEM_NUMBER 列的 UNIQUE 约束是 DEFERRABLE。如果想要确保在每个货品号之间有至少 500 的数字间隙，可执行以下 UPDATE 语句：

```
UPDATE inventory SET item_number = item_number + 500
```

但是，如果在任何一个 ITEM_NUMBER 列的当前值上加 500 会引起向该列插入重复的值，则 DBMS 会中止语句的执行（如果已有的 ITEM_NUMBER 值为 101 和 601，是这种情况）。但是，如果 DBMS 是在更新了所有货品号之后才检查 UNIQUE 约束，则不会出现重复的值。

如果执行以下 SET CONSTRAINTS 语句：

```
SET CONSTRAINTS ALL DEFERRED
```

DBMS 将把会话中的其余部分所有的 DEFERRABLE 约束设置为 DEFERRED。结果，下面的 UPDATE 语句将能执行而没有错误：

```
UPDATE inventory SET item_number = item_number + 500
```

因为 DBMS 将不对 ITEM_NUMBER 检查 UNIQUE 约束，直到在更新了 INVENTORY 表中的所有行之后执行了隐含的 COMMIT 语句时为止。

要理解的重要事情是，只能将约束延缓到提交（COMMIT）事务处理时。到那时，DBMS 总是检查所有已延缓的约束，从而确保没有事务处理工作违反数据库的完整性。如果想要做的工作要求系统延缓不止一条 SQL 语句的约束检查，那么既可将这些语句放在一起作为单条批处理来执行，也可明确地执行 BEGIN TRANSACTION（或 BEGIN TRAN）语句，此后必须正常地结束会话或是执行 COMMIT 语句来结束事务处理。

除了允许立刻延缓系统检查所有 DEFERRABLE 约束的时间之外，SET CONSTRAINTS 语句的如下句法还允许指定单独的命名约束，其约束检查为 IMMEDIATE，而不是想要延缓的：

```
SET CONSTRAINTS
(<constraint_name>|ALL) {DEFERRED|IMMEDIATE}
```

例如，如果想要对 ITEM_NUMBER 列仅延缓 NON_ZERO_COST 约束而不延缓 UNIQUE 约束，可执行下面的 SET CONSTRAINTS 语句：

```
SET CONSTRAINTS non_zero_cost DEFERRED
```

这就将 NON_ZERO_COST 约束指定为在提交事务处理时要检查的约束。

注意：由于 SET CONSTRAINTS 语句改变了会话中其余语句的约束检查时间，在提交（COMMIT）了需要延缓约束检查的事务处理之后，一定要执行以下语句：

```
SET CONSTRAINTS ALL IMMEDIATE
```


第 15 章 编写外部应用程序来查询与操作数据库数据

技巧 276 为开放数据库互连 (ODBC) 连接创建数据源名称 (DSN)

顾名思义, 数据源名称 (data source name, DSN) 就是运行在计算机上的程序所能了解的数据源的名称。当把 DSN 与 DBMS (如 MS-SQL Server、Oracle 或 DB2) 连接起来之后, 应用程序使用“开放数据库连接性”(Open Database Connectivity, ODBC) 驱动程序通过名称与 DBMS 连接起来并向其发送 SQL 语句。ODBC 驱动程序不仅将 SQL 语句传递给 DBMS 以便执行, 而且还向程序返回查询结果。

当在可访问的网络工作站、服务器或独立的个人计算机上安装并启动了 SQL 服务器之后, 就可通过执行以下步骤为 DBMS 创建 DSN 了:

1. 选择 Windows 的 Start 菜单上的 Settings 选项, 再单击 Control Panel, 从而打开 Control Panel (控制面板)。
2. 双击 ODBC Data Sources (ODBC 数据源) 图标。Windows 将启动 ODBC 管理器, 显示 ODBC Data Source Administrator (ODBC 数据源管理器) 对话框, 如图 15.1 所示。

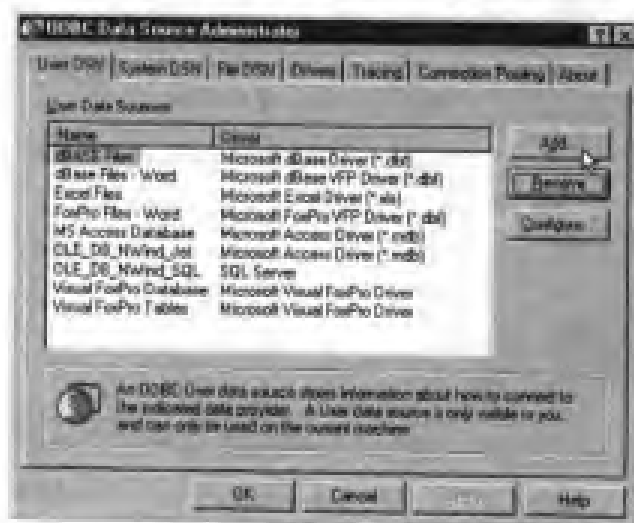


图 15.1 ODBC Data Source Administrator 对话框的 User DSN (用户数据源名称) 选项卡

3. 单击 Add (添加) 按钮。ODBC 管理器将启动 Create a New Data Source Wizard (创建新数据源向导), 提示用户选择在与数据库通讯时的驱动程序。

4. 为了使用与 SQL 服务器通讯的 ODBC 驱动程序, 单击 Create a New Data Source Wizard 上的滚动框中的 SQL Server。然后单击 Finish 按钮。向导将提示使用如图 15.2 所示的对话框来描述 SQL 服务器。

5. 在 Name 域中键入当需要与 SQL 服务器通讯时想要应用程序使用的数据源名称 (DSN)。所键入的名称不必是任何特定的名称。例如, 可以使用 SQL 服务器的名称或是基于服务器的 DSN 的默认数据库的名称。对本例来说, 可键入 MSSQLServer。

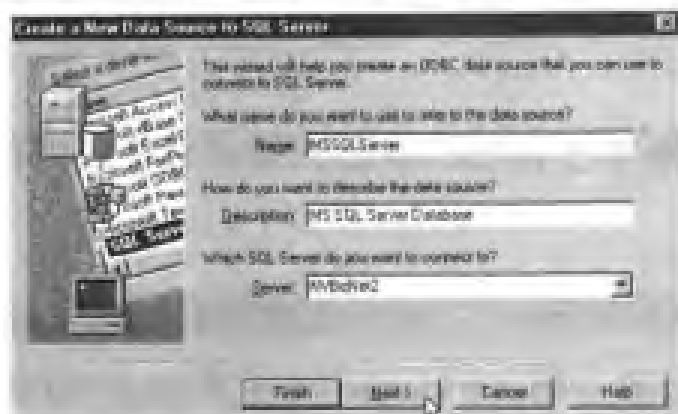


图 15.2 Create a New Data Source Wizard 的数据库描述屏幕

注意：如果打算为多个 SQL 服务器建立 DSN，最好在每个 DSN 中包括服务器名。如果这样做了，当想让程序与特定的服务器通讯时，每个名称可表明应使用哪个 DSN。例如，如果有一台在名为 NVBIZNET2 的 NT 服务器上运行的 MS-SQL Server，应在 Name 域中键入类似于 MSSQLServer_NVBizNet2 一类的具有描述性的名称。

6. 在 Description 域内键入数据源的描述。对本例来说，可键入 Microsoft SQL Server Database。

7. 在 Server 域内键入 DBMS 正运行其上的服务器的名称。如果 DBMS 运行于定义 DSN 的计算机上，可单击 Server 域右边的下拉列表按钮并选择(local)。

8. 单击 Next 按钮。ODBC 将提示输入用户名和密码信息，如图 15.3 所示。

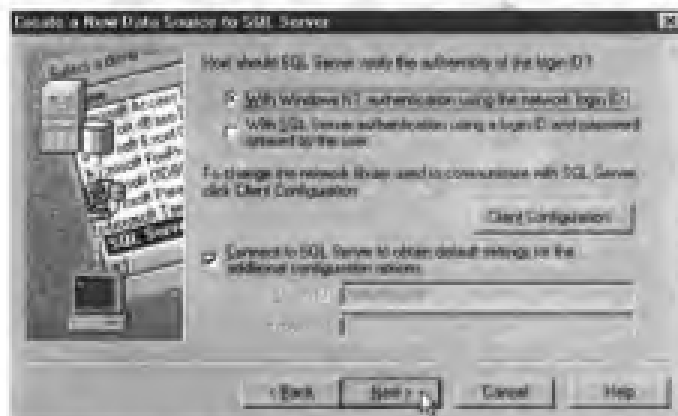


图 15.3 Create a New Data Source Wizard 的用户名/密码屏幕

9. 如果 MS-SQL Server 使用 Windows NT Integrated Security（综合安全性，此内容已在技巧 98“理解 MS-SQL Server 标准和 Windows NT 综合安全性”中学习过），可接受如图 15.3 所示的默认用户名和密码。服务器将假设 Windows NT 在登录时认证用户名和密码对。为了让 SQL Server 认证用户名和密码对，可单击 With SQL Server Authentication Using a Login ID and Password Entered by the User 左边的单选钮，然后分别在 Login ID 和 Password 域内键入 MS-SQL Server 用于认证的用户名和密码。

10. 单击 Next 按钮。向导试图与在步骤 7 中指定的 SQL 服务器连接。如果 SQL 服务器还没有运行，或者如果 DBMS 不认识在步骤 9 中键入的用户名和密码对，ODBC 将挂起一段时间（小于一分钟），然后显示错误信息，其中解释了为什么连接企图失败。为了创建 DSN，会

话必须能够与指定的 DBMS 连接。因而，采取必要的改正措施，然后重复步骤 10。

11. 在使用步骤 9 中指定的用户名/密码对与步骤 7 中选定的 DBMS 连接之后，向导将提示使用图 15.4 所示的对话框来输入 SQL 的默认数据库。

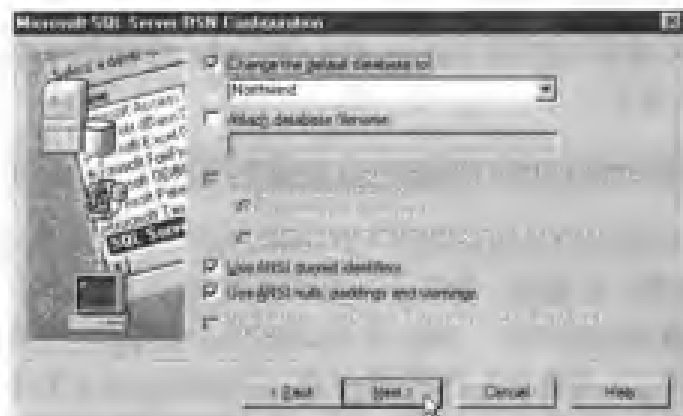


图 15.4 Create a New Data Source Wizard 的默认数据库（第 4 个）屏幕

12. 为了使用所创建的与 DBMS 连接的 DSN 为应用程序选择默认数据库，单击 Change the Default Database To（将默认数据库改变为）标签左边的复选框使之出现选择标记。

13. 单击默认数据库名域左边的下拉列表按钮并从下拉列表中选择默认数据库。对本例来说，可单击 Northwind，然后单击 Next 按钮。

14. 根据运行在计算机上的 ODBC 的版本的不同，向导还将显示一个到两个设置屏幕。只要单击 Next 按钮接受每个屏幕的默认设置，直到到达向导的最后屏幕，在那里可单击 Finish 按钮。向导接着显示。如图 15.5 所示的 ODBC Microsoft SQL Server Setup 对话框。

15. 单击 Test Data Source（测试数据库源）按钮。向导将使用在步骤 5~步骤 13 中输入的设置试图与 DBMS 连接。如果测试不成功，则返回 DSN 设置步骤并持之以按向导错误信息中的要求作出必要的改正，然后重复步骤 15。在向导指出成功完成数据源测试之后，单击 Test Results（测试结果）对话框上的 OK 按钮。

16. 单击 ODBC Microsoft SQL Server Setup 对话框上的 OK 按钮。



图 15.5 Create a New Data Source Wizard 的测试 ODBC 设置（最后一个）屏幕

在完成步骤 16 之后，Create a New Data Source Wizard 将在计算机上注册新的 DSN 并在

ODBC Data Source Administrator 对话框上的 User Data Sources（用户数据源）列表框中显示其名称和描述。为了退出 ODBC Data Source Administrator 应用程序，可在 ODBC Data Source Administrator 对话框底部的一行按钮中单击 OK 按钮（左边的第一个按钮）。

注意：为了使 DSN 对连接到机器上的所有用户都是可见的（而不是只对创建 DSN 时的用户名可见），应将 DSN 添加到 Data Source Administrator 对话框的 System DSN 选项卡，而不应添加到 User DSN 选项卡。

技巧 277 向 Visual Basic (VB) 窗体中添加数据控件组件以便提取 SQL 表数据

Visual Basic (VB) 的数据控件组件提供 VB 应用程序可以使用的一些子程序（方法），用来处理保存在 SQL 数据库中的数据。事实上，在向 VB 窗体中添加数据控件组件之后，应用程序就能够执行大多数数据访问任务，而完全不必编写任何代码。另外，如果将 MSFlexGrid 控件绑定到数据控件组件上（读者将在技巧 278“向 Visual Basic (VB) 窗体中添加 MSFlexGrid 控件以显示 SQL 表数据”中学习有关内容），则 VB 程序将具有提取、操作并显示来自 SQL 服务器上的对其有访问权限的一个或多个表中的数据的工具。

为了向 VB 窗体添加数据控件组件，请执行以下步骤：

1. 如果还没有做这项工作，可启动 MS-Visual Basic (VB) 并打开一个 Standard .EXE 工程。
2. 双击如图 15.6 所示的 VB 窗口沿左边的 VB 工具箱上的 Data 图标。VB 将向当前活动窗体（在本例中是 Form1）上添加数据控件组件。
3. 使用鼠标将数据控件拖向窗体的左下角（在完成当前工程的时候，此工程将跨过技巧 277 和技巧 278，读者将把数据控件放在 MSFlexGrid 控件之下并调节其尺寸）。
4. 为了处理数据控件（Data Control）的属性，可选择 View 菜单中的 Properties Window 选项（或按 F4 键）。

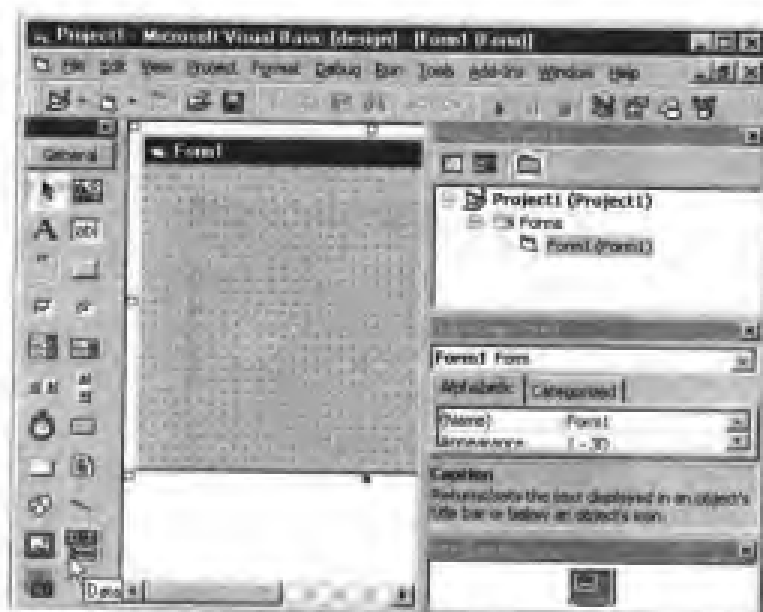


图 15.6 Microsoft Visual Basic 5.0 应用程序窗口

5. 单击 Caption 属性标签（数据控件的属性标签[及其属性]是按字母顺序排列在属性窗口的 Alphabetic 选项卡中的）。

6. 单击 Caption 属性域（紧靠 Caption 属性标签的右边）并键入想要 VB 显示在数据控件上的标签。对本例来说，可在 Caption 属性域中键入 Product/Supplier。

7. 单击 DatabaseName 属性标签。然后单击 DatabaseName 属性域并键入其中有想要访问的数据库的 SQL 服务器的数据源名称（DSN）。如果所要的数据库位于在技巧 276 “为开放数据库互连（ODBC）连接创建数据源名称（DSN）”中为其创建（并注册）DSN 的 SQL 服务器上，可在 DatabaseName 属性域中键入 MSSQLServer。

8. 单击 Connect 属性标签。接着单击 Connect 属性域并键入 ODBC Driver Manager 与 SQL 服务器连接所需的信息，其形式为：

```
ODBC;UID=<username>;PWD=<password>;DATABASE=<database name>
```

例如，为了以用户 KONRAD 及密码与 KING 与 SQL 服务器连接并使用 NORTHWIND 数据库，可在 Connect 属性域中键入：

```
ODBC;UID=KONRAD;PWD=KING;DATABASE=NORTHWIND
```

一定要用自己的用户名和密码来代替本例中的用户名 KONRAD 和密码 KING。

注意：如果 DSN 默认值就是为想要使用的数据库，或者，如果 DBA 已经将其指定为默认的数据库时，则可在连接字符串中忽略 DATABASE=<database name> 部分。

当 VB 应用程序使用数据控件组件的方法来访问数据库时，Windows 将把数据控件的连接属性与 DSN 的连接设置组合起来并试图建立与 SQL 服务器的连接。如果忽略了用户名或密码（或者如果在 Connect 属性域中键入的用户名/密码对是非法的），ODBC 驱动程序将提示用户输入适当的用户名/密码对，以便登录到带有在步骤 7 中键入的 DSN 的 SQL 服务器上。

技巧 278 向 Visual Basic (VB) 窗体中添加 MSFlexGrid 控件以显示 SQL 表数据

MSFlexGrid 控件给予 VB 应用程序以显示和处理表状数据的能力。使用 MSFlexGrid 控件的方法（子程序），VB 程序可以对包括文本或图形图像的表进行排序、合并以及格式化等操作。如果将 MSFlexGrid 控件绑定到数据控件上（读者在技巧 277 “向 Visual Basic (VB) 窗体中添加数据控件组件以便提取 SQL 表数据”的工程中添加到 VB 窗体的控件），可将其用于显示数据控件从 SQL 服务器上提取的表中的只读数据。

为了向 Visual Basic 窗体中添加 MSFlexGrid 控件，可执行以下步骤：

1. 如果还没有添加，可启动 MS-Visual Basic (VB)，打开一个 Standard .EXE 工程，根据技巧 277 中的步骤向窗体添加数据控件。

2. 为了向工程中添加 MSFlexGrid 控件组件，可右击工程的工具箱（沿 VB 应用程序窗口的左边）并从弹出菜单中选择 Components（组件）命令。接着，VB 将显示如图 15.7 所示的 Components 对话框。

3. 使用 Controls 选项卡上的 Components 列表框右边的滚动条滚动控件组件列表，直到看到 MSFlexGrid 控件。接着单击 Microsoft FlexGrid Control 5.0 左边的复选框使之出现选择标记（选择系统中 MSFlexGrid 控件可用的最高版本）。

4. 单击靠近 Components 对话框中部的 OK 按钮。VB 将返回 VB 应用程序窗口并把 MSFlexGrid 控件作为最新控件组件添加到 VB 工具箱中。

5. 双击 MSFlexGrid 图标向 VB 窗体中添加 MSFlexGrid 控件。

6. 为了使用 MSFlexGrid 控件的属性，选择 View 菜单中的 Properties Window 选项（或按

F4 键)。

7. 单击 DataSource 属性标签 (MSFlexGrid 控件的属性标签[及属性]在 Properties 窗口的 Alphabetic 选项卡上以字母顺序列出)。

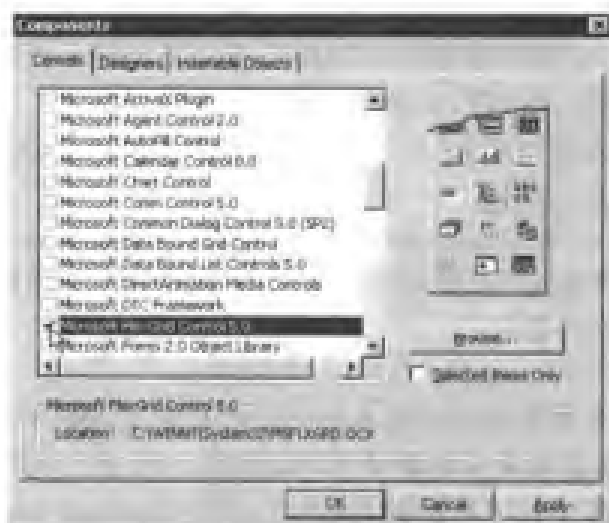


图 15.7 Microsoft Visual Basic 5.0 的 Components 对话框

8. 单击 DataSource 属性域右边的下拉列表按钮 (紧靠 DataSource 属性标签的右边) 并选择将向 MSFlexGrid 控件提供数据的数据控件名称。对本例来说, 在技巧 277 中已向 VB 窗体中添加了名为 Data1 的单个数据控件。因而从下拉列表中选择 Data1。

9. 右击 MSFlexGrid 控件 (在 VB 窗体上) 并从弹出的菜单中选择 Properties 命令。VB 将显示 Properties Pages (属性页) 对话框。

10. 设置想要 MSFlexGrid 控件显示数据的行数和列数。使列数等于所期望的 SQL 查询返回的列数。对本例来说, 在 Properties Pages 对话框的 General 选项卡上的 Rows 域内键入 6 而在 Columns 域内键入 4。

11. 在 Fixed Cols (固定列) 域内键入 MSFlexGrid 控件要显示的标题列和行的数目。对本例来说, 在 Fixed Rows (固定行) 域内键入 1 而在 Fixed Cols (固定列) 域内键入 0。(读者不必键入列标题, VB 将在运行时填入)。

12. 为了让用户在 VB 应用程序运行时重新设置列的尺寸, 可单击 AllowUserResizing 域右边的列表框并从下拉列表中选择 1-Columns。

13. 单击 Properties Pages 对话框底部的 OK 按钮。VB 将返回 VB 设计应用程序窗口。

完成步骤 13 之后, VB 窗体将包括绑定到数据控件 (名为 Data1, 是在技巧 277 中添加到窗体上的) 上的 MSFlexGrid 控件。

现在, 使用鼠标来排列控件并重设其尺寸 (在每个控件的改变尺寸控制柄上使用鼠标指针), 使得 VB 窗体看起来类似于图 15.8 的样子。



图 15.8 带有 MSFlexGrid 控件 (在上部) 和数据控件 (靠近底部) 的 VB 窗体

技巧279 向 Visual Basic (VB) 窗体中添加 Text 和 Button 控件创建向 SQL Server 发送查询的应用程序

在技巧 277 “向 Visual Basic (VB) 窗体中添加数据控制组件以便提取 SQL 表数据”中, 读者学习了如何在 VB 窗体上放置数据控件, 以便为应用程序提供从 SQL Server 上提取数据所需的方法 (子程序调用)。接着, 技巧 278 “向 Visual Basic (VB) 窗体中添加 MSFlexGrid 控件以显示 SQL 表数据”向读者展示了如何添加 MSFlexGrid 控件以便显示所提取的数据。现在只需要添加允许用户键入 SQL 查询的文本框控件以及告诉 VB 向 SQL Server 发送查询的按钮控件, 读者就有了可查询 SQL DBMS 并显示其数据的 VB 应用程序。

注意: 本技巧的剩下部分假设读者当前已有以设计模式打开的 VB, 其中有一个单窗体的工程, 并用技巧 277 的步骤添加了数据控件, 用技巧 278 的步骤添加了 MSFlexGrid 控件。

为了向 VB 窗体上添加文本和按钮控件, 请执行以下步骤:

1. 双击 VB 工具箱中的 TextBox 图标 (如果不认识 TextBox 图标, 可将鼠标指针放在 VB 工具箱的每个图标上, 让 VB 显示图标所代表的控件名称)。
2. 为了使用 TextBox 控件的属性, 请选择 View 菜单中的 Properties Window 选项 (或持之以恒按 F4 键)。
3. 单击 (Name) 属性标签 (TextBox 控件的属性标签 [及属性] 在 Properties 窗口的 Alphabetic 选项卡上以字母顺序列出)。
4. 单击 (Name) 属性域 (紧靠 (Name) 属性标签的右边)。对本例来说, 在 (Name) 属性域中键入 SupplierName。
5. 单击 Text 属性标签。接着单击 Text 属性域并删除其内容。
6. 为了向 VB 窗体添加 CommandButton 控件, 可双击 VB 工具箱中的 CommandButton 图标。
7. 为了处理 CommandButton 控件的属性, 选择 View 菜单中的 Properties Window 选项 (或持之以恒按 F4 键)。
8. 单击 (Name) 属性标签。接着单击 (Name) 属性域并键入当鼠标单击按钮时所引用的名称。对本例来说, 在 (Name) 属性域中键入 SearchButton。
9. 单击 Caption 属性标签。接着单击 Caption 属性域, 并键入想要 VB 放到按钮上的标签。对本例来说, 在 Caption 属性域键入 Search。
10. 重复步骤 6~步骤 9, 而此次在步骤 8 中的 (Name) 属性域中键入 CloseButton, 而在步骤 9 中的 Caption 属性域中键入 Close。

在完成步骤 10 之后 (第二次), 对 VB 窗体上的两个 CommandButton 控件和 TextBox 控件改变位置和尺寸, 如图 15.9 所示。



图 15.9 VB SQL 查询窗体, 其中有数据控件、MSFlexGrid 控件、TextBox 控件和两个 CommandButton 控件

既然在 VB 窗体上有了所有控件,剩下要做的事就是编写想要计算机在用户单击每个命令按钮时执行的代码。例如,假设想要 Search 按钮发送以下 SELECT 语句:

```
SELECT productid 'Product ID', productname 'Description',
companyname 'Supplier', s.supplierid 'ID'
FROM products p, suppliers s
WHERE p.supplierid = s.supplierid
AND companyname LIKE '%:SupplierName%'
ORDER BY supplier, description
```

这将列出由供货商(其名称或部分名称是用户在窗体的 SupplierName[这是一个 TextBox 控件]域内向 SQL 服务器键入的)提供的所有产品(来自 PRODUCTS 表)的清单。

为了将代码添加到 SEARCH 按钮,可双击 SEARCH 按钮打开 VB 的代码窗格并显示 SearchButton 方法(子程序),然后如图 15.10 所示键入 VB 代码。

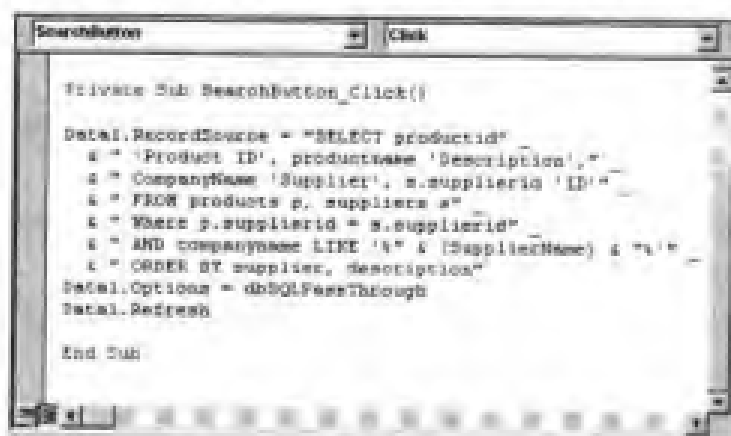


图 15.10 VB 应用程序的代码窗格,其中有名为 SearchButton 的 CommandButton 控件

在快速地检验 Search 按钮的代码时,请考虑以下因素:

- Data1 是在技巧 277 中为数据控件起的名称。
- 想要让 SQL 服务器执行的 SQL 语句必须放在数据控件的 RecordSource 属性中。
- 在数据控件的 Options 属性中放入枚举的常数 dbSQLPassThrough,告诉 Jet 数据库引擎不要分析查询或检查其句法,只是将其传递到 SQL 服务器以备处理。
- 数据控件的 Refresh 方法调用将 RecordSource 属性中的查询发送到 SQL 服务器,然后在组成 MSFlexGrid 控件的行和列中接受并显示查询结果。

选择 View 菜单中的 Object (对象) 选项重新在 VB 应用程序窗口显示 VB 窗体。接着双击有 Close 标签的 CommandButton 控件,以编辑在用户单击 Close 按钮时 VB 将执行的代码。

对本工程来说,在 VB 应用程序窗口的代码窗格中键入:

```
Private Sub CloseButton_Click()
    Unload Me
End Sub
```

接着,选择 View 菜单中的 Object (对象) 选项重新显示所创建的 VB 窗体。

最后,选择 Run (运行) 菜单中的 Start (开始) 选项,测试一下 VB 应用程序。在 VB 解释器在屏幕上显示窗体之后,在窗体的文本框中键入供货商名,然后单击 Search 按钮。例如,为了显示由名称中带 new 的公司提供的所有产品,可在文本框中键入 new,再单击 Search 按钮。VB

应用程序（刚刚编写的）将以显示产品和供货商（Products/Supplier）加以响应，如图 15.11 所示。



图 15.11 VB 应用程序窗口，其中有 Northwind 数据库的 SQL 查询结果，供货商名称中有 new

为了顺利地结束 VB 应用程序，可单击关闭按钮。

技巧 280 创建用于与 SQL Server 通讯的简单 C++ 外壳程序

在写作本书时，C 仍然是编写 Windows 应用程序的首选语言，特别是那些与 MS-SQL Server 以及其他 DBMS 通讯的程序更是如此。但是，正如读者在技巧 277~技巧 279 中所看到的，Visual Basic 与数据源（如 SQL 服务器）的（几乎）无缝接合和易于使用在最近的将来必定成为 C 语句的强大竞争对手。本技巧向读者展示如何编写 C 的主要程序外壳，可用于技巧 281~技巧 291 中与 SQL 服务器连接与通信。

当在 Windows 环境中编写程序时，以下的 MS-DOS 的“Hello World!”程序：

```
#include <stdio.h>
void main()
{ printf ("Hello world!\n");
  return; }
```

变为：

```
#include <windows.h>
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int iCmdShow)
{
  MessageBox (0, "Hello World!", "1001 SQL Tips - Tip 372",
              MB_OK);
  return 0;
}
```

程序中的第一条语句（`#include <windows.h>`）告诉编译器包括 WINDOWS.H 头文件，这是每一种为 Windows 编程的 C 编程环境中都有的。接着，WINDOWS.H 中又有 `#INCLUDE` 语句（形式为 `#include <filename>.h`），这就告诉编译器包括另外的头文件，这些头文件中包括 Windows 函数、结构、数据类型和数值常数的声明。

每种 C 程序都有一个程序开始执行的“主”入口点。当 Windows 启动编译好的程序（.EXE 文件）时，系统首先执行由编译器插入的几行“启动”代码。编译器的启动代码接着调用由程序员编写的 `main` 函数。对于 MS-DOS 程序来说，主入口点就是 `main()` 函数。对于 Windows 程序来说，主入口点总是称为 `WinMain`。在本例中，“Hello World!”程序使用 `WINAPI` 调用序

列，在程序结束时向 Windows 操作系统返回整数（int）数据类型的值（程序中的最后的语句 [return 0;]告诉主函数[WinMain]，在程序完成其执行之后，向 Windows 返回值 0）。

Windows 向“Hello World!”程序传递的第一个参数 hInstance 是实例句柄，是向 Windows 操作系统识别程序的惟一的数。可把 hInstance 看作程序的进程 ID（或 PID），诸如 Windows NT 这样的多任务操作系统指定给每个并发执行的程序（或任务）的惟一的数。如果启动同一程序的多个副本（实例），Windows 为每个实例指定一个惟一的实例句柄，然后使用 hInstance 参数向应用程序的 WinMain 例程中传递句柄的值。

第二个参数 hPrevInstance，在 Windows 95 下已变得有点过时，是为了向后兼容性而保留下来的。如果 Microsoft 消除了第二个参数，运行在 Windows/NT 平台上的所有 C 程序必须重新编译并重新发布。（你能想像这一任务的艰巨性吗？）hPrevInstance 是程序的当前实例之前的最近仍然活动的副本的句柄。如果没有其他的副本在运行，Windows 将 hPrevInstance 设置为 0 或 NULL。从 Windows 95 和目前的 NT 以来的 Windows 版本都将 hPrevInstance 设置为 NULL，而不管当前运行在会话中的程序副本的数目有多少。

lpCmdLine 是一个指向以 NULL 结束的字符串的长（32 位）指针，其中包括 Windows 要传递给程序的任意命令行参量。例如，如果在 MS-DOS 命令行或是 Start 菜单的 Run 选项对话框中键入以下内容启动程序 TIP372.EXE：

```
TIP372 Parameters for startup
```

Windows 将把一个指向保存着字符串 Parameters for startup 的内存位置的指针传递给程序。

最后一个参数 iCmdShow，告诉应用程序用户想让 Windows 如何显示程序运行的会话窗口。iCmdShow 参数的不同值列在 WINRESRC.H 头文件（此文件已经由 #INCLUDE 语句包括在 WINDOWS.H 头文件中了）的 ShowWindow()命令部分。通常，iCmdShow 有值为 1（SW_SHOWNORMAL），使程序运行在显示于 Windows 桌面的活动的正常窗口中，或者值为 7（SW_SHOWMINNOACTIVE），使程序运行在最小化的任务栏图标。

对示例程序中惟一做实际工作的语句不用管得太多：

```
MessageBox (0, "Hello World!", "1001 SQL Tips - Tip 372",  
MB_OK)
```

Windows 的 MessageBox 函数只是在屏幕上以 Windows 消息框来显示“Hello World!”消息。随着读者 C 程序写得越多，不得不记住常常使用的函数所需的参数。在（几乎）所有情况下，当调用一个函数时，不用了解函数是如何做其工作的，而只需要了解函数可做什么（而不是如何做）以及在调用函数时必须向其传递的参数。如果进行技巧 281~技巧 291 中的工程，则将建立起可用作本技巧示例的基本的 C 程序外壳，最后可得到一个可用于访问任何 SQL DBMS 的工作程序，只要有其 ODBC 驱动程序并定义了数据源名称（DSN）即可。

注意：C 是大小写敏感的，因此，函数 WinMain 与 winmain 或 WINMAIN 是不同的。

技巧 281 使用 SQLAllocEnv 和 SQLFreeEnv 分配及释放 ODBC 环境资源

开放数据库连接（Open Database Connectivity, ODBC）是一套函数（应用程序编程接口 [API]调用），程序可用来处理任何有 ODBC 驱动程序的数据库中的数据。由于只必须了解哪个函数调用执行什么任务，ODBC 调用级的接口使得程序员不必了解想要用于提取、更新并存储其数据的每种 DBMS 的细节。程序员只需通过调用适当的 ODBC 函数指明想要 DBMS

执行的功能（或工作）。接着函数让特定的为想要访问 DBMS 的 ODBC 驱动程序来处理执行请求所涉及的细节。

在调用任何其他 ODBC 函数之前，应用程序必须调用 SQLAllocEnv 来为 ODBC 环境句柄分配内存并初始化 ODBC 调用级的接口。请注意，在以下示例程序中传递到 SQLAllocEnv 的参数是内存地址，该处是函数保存程序的 ODBC 环境的实际物理地址（句柄）：

```
#INCLUDE <stdio.h>
#include <windows.h>
#include <sqlext.h>
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int iCmdShow)
{
    HENV henv;                //pointer to a memory location
    RETCODE retcode;           //signed short (16-Bit) integer
    LPSTR retcode_text = " ";  //character string
    retcode = SQLAllocEnv(&henv); //allocate environment handle
    if (retcode == SQL_SUCCESS)
    {
        /* CONNECT TO SQL SERVER & PERFORM WORK HERE */
        SQLFreeEnv(henv); //free environment handle & its memory
    }
    else
    {
        sprintf(retcode_text, "Error on SQLAllocEnv = %d",
                retcode);
        /* display any error messages */
        if (strcmp(retcode_text, " ") != 0)
            MessageBox (0, retcode_text, "Connect to MSSQLServer",
                        MB_OK|MB_ICONERROR);
        return 0;
    }
}
```

在本例中，如果 SQLAllocEnv 在分配和初始化 ODBC 环境所使用的内存区域时是成功的，指针 henv 将包括应用程序的函数所使用的 ODBC 环境的内存地址（或句柄）（例如，在技巧 282 “使用 SQLAllocConnect 和 SQLFreeConnect 分配和释放连接句柄和内存资源”中，当调用 SQLAllocConnect 函数初始化 ODBC 环境的连接部分时，将要把 ODBC 环境句柄传递给 SQLAllocConnect 函数）。

请注意，在本例中的 C 程序告诉编译器（除了包括 STDIO.H 和 WINDOWS.H 之外）包括 SQLEXT.H 头文件。SQLEXT.H 头文件接着又包括类型定义和告诉编译器包括 SQL.H、SQLTYPES.H 和 SQLUCODE.H 等头文件的 #INCLUDE 语句。4 个头文件一起为 C 应用程序提供访问 SQL 服务器中的数据所需的数据类型、函数调用、结构和常数。

技巧 282 使用 SQLAllocConnect 和 SQLFreeConnect 分配和释放连接句柄和内存资源

SQLAllocConnect 和 SQLFreeConnect 在功能上分别与 SQLAllocEnv 和 SQLFreeEnv 是类似的。当程序建立与 SQL 服务器的连接时，ODBC 驱动程序必须跟踪有关会话的几个细节。

例如，由于发送到 SQL 服务器的每条命令必须由 DBMS 的安全机制加以确认，ODBC 驱动程序将每个连接的用户名和密码保存在文件中。利用这一方法，当把语句发送到服务器执行时，驱动程序可将用户名和密码对一起传递到 DBMS。类似地，ODBC 驱动程序必须跟踪 DBMS 在会话中正在执行的任何语句以及会话是否有任何打开的事务处理。

ODBC 驱动程序将有关每个连接所需的信息保存在 ODBC 环境空间部分（即内存区域）内，这部分内存区域是通过调用 SQLAllocEnv 函数分配的（在技巧 281 “使用 SQLAllocEnv 和 SQLFreeEnv 分配及释放 ODBC 环境资源”中分配的）。虽然 SQLAllocEnv 函数将一部分计算机总的内存映射出来为应用程序和 DBMS 之间的 ODBC 驱动程序接口所用，但 SQLAllocConnect 函数却将由 SQLAllocEnv 分配的作为存储有关每个 ODBC 连接细节所用的部分内存再映射出来。当调用 SQLAllocConnect 时，函数不仅映射出部分 ODBC 环境空间，而且函数还返回一个指针（称为连接句柄），这个指针给出连接设置区域在内存中的 ODBC 环境区域的起始地址。

SQLAllocConnect 函数调用的句法如下：

```
RETCODE SQLAllocConnect(henv, phdbc)
```

其中：

- henv 是环境句柄，也就是指向程序可找到由 SQLAllocEnv 函数（读者在技巧 281 中学习过有关内容）分配给 ODBC 环境所使用的内存位置的指针。
- phdbc 是数据库连接句柄，也就是指向程序可找到特定的与 DBMS 连接的设置和信息的内存位置的指针。

在技巧 281 的示例程序中，也可使用以下语句调用 SQLAllocConnect 来建立连接句柄：

```
retcode = SQLAllocConnect(henv, &hdbc);
```

请注意（正如 SQLAllocEnv 的情况一样），第二个参数是 hdbc（指针）变量的内存地址（在程序中用 HDBC hdbc; 语句所定义的）。SQLAllocConnect 函数将把实际的连接区域（连接句柄）放在 hdbc 变量中。

正如应用程序应该调用 SQLFreeEnv 函数来释放由 SQLAllocEnv 分配的不再需要的内存一样，程序也应该调用 SQLFreeConnect 函数来释放由 SQLFreeConnect 所分配的不再需要的内存。

SQLFreeConnect 函数调用的句法如下：

```
RETCODE SQLFreeConnect(hdbc)
```

其中：

- hdbc 是由调用 SQLFreeConnect 函数分配的连接区域的连接句柄（指向起始内存位置的指针）。

如果添加了 SQLAllocConnect 和 SQLFreeConnect 函数，在技巧 281 中开始的 C 程序将变成下面的样子：

```
#INCLUDE <stdio.h>
#include <windows.h>
#include <sqlext.h>
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int iCmdShow)
{
```

```

HENV henv;                //HENV is data type: void*
HDBC hdbc;                //HDBC is data type: void*
RETCODE retcode;          //signed short (16-Bit) integer
LPSTR retcode_text = " "; //character string
retcode = SQLAllocEnv(&henv);
if (retcode == SQL_SUCCESS)
{
    //allocate connection handle
    retcode = SQLAllocConnect(henv,&hdbc);
    if (retcode == SQL_SUCCESS)
    {
        /* CONNECT TO SQL SERVER & PERFORM WORK HERE */
        SQLFreeConnect(hdbc);
    }
    else
        sprintf(retcode_text,"Error on SQLAllocConnect = %d",
            retcode);
    SQLFreeEnv(henv);
}
else
    sprintf(retcode_text,"Error on SQLAllocEnv = %d",
        retcode);
/* display any error messages */
if (strcmp(retcode_text," ") != 0)
    MessageBox (0, retcode_text, "Connect to MSSQLServer",
        MB_OK|MB_ICONERROR);
return 0;
}

```

技巧 283 使用 SQLSetConnectOption 为与 SQL Server 的 ODBC 连接设置会话选项

在调用 SQLAllocConnect 函数分配连接句柄和 DBMS 连接使用的内存区域之后, 可使用 SQLSetConnectOption 函数来设置连接选项, 这些选项都已总结在表 15.1 中了。SQLSetConnectOption 函数调用的句法如下:

```
RETCODE SQLSetConnectOption(hdbc, wOption, dwOpVal)
```

表 15.1 SQLSetConnectionOption 函数选项和选项值

wOption (连接选项)	dwOpVal (连接选项设置)
SQL_ACCESS_MODE	SQL_MODE_READ_ONLY、SQL_MODE_READ_WRITE
SQL_AUTOCOMMIT	SQL_AUTO_COMMIT_OFF、SQL_AUTO_COMMIT_ON
SQL_CURRENT_QUALIFIER	一个包括数据源限定符的 null 结尾的字符串。在 MS-SQL Server 上, 数据源限定符是数据库名称。因此, 驱动程序将向 DBMS 发送 USE <database> 语句, 其中 <database> 是在 dwOpVal 中提供的字符串
SQL_LOGIN_TIMEOUT	等待 DBMS 完全登录请求的秒数。值为 0 时禁用时限, 驱动程序将无限地等待下去, 直到 DBMS 完成登录请求
SQL_ODBC_CURSORS	SQL_CUR_USE_IF_NEEDED、SQL_CUR_USE_ODBC、SQL_CUR_USE_DRIVER

续表

wOption（连接选项）	dwOpVal（连接选项设置）
SQL_OPT_TRACE	SQL_OPT_TRACE_OFF、SQL_OPT_TRACE_ON
SQL_OPT_TRACEFILE	以 NULL 结尾的字符串形式表示的跟踪文件名
SQL_PACKET_SIZE	网络封包容量，以字节计
SQL_QUIET_MODE	ODBC 驱动程序显示对话框的窗口句柄。如果设置为等于 NULL 指针，则 ODBC 驱动程序将不显示任何对话框
SQL_TRANSLATE_DLL	一个 null 结尾的字符串，其中有包括函数 SQLDriverToDataSource 和 SQLDataSourceToDriver 的 ODBC 驱动程序要装入并用来执行诸如字符串集翻译的 DLL 文件名
SQL_TRANSLATE_OPTION	ODBC 驱动程序传递给翻译 DLL 的 32 位的整数值
SQL_TXN_ISOLATION	SQL_TXN_READ_UNCOMMITTED、SQL_TXN_READ_COMMITTED、SQL_TXN_REPEATABLE_READ、SQL_TXN_SERIALIZABLE、SQL_TXN_VERSIONING

其中：

- hdbc 是由 SQLAllocConnect 返回的 ODBC 环境空间（内存）的连接区域的连接句柄（指向内存起始位置的指针）
- wOption 是表 15.1 中的连接选项之一。
- dwOpVal 是为 wOption 设置的值。根据调用 SQLSetConnectOption 函数要设置的选项，dwOpVal 既可是 32 位的整数也可是 NULL 结尾的字符串。

程序必须为每个想要设置的会话连接选项调用 SQLConnectOption 函数一次。

当通过执行下面的语句调用 SQLSetConnectOption 时：

```
retcode = SQLSetConnectOption(hdbc, SQL_AUTOCOMMIT,
SQL_AUTOCOMMIT_ON);
```

ODBC 驱动程序为所有活动的语句和所有的接下来使用 ODBC 函数调用发送到 DBMS 处理的语句设置选项（在技巧 284“使用 SQLConnect 和 SQLDisconnect 建立和结束 DBMS 会话”中的代码示例使用了 SQLSetConnectionOption 函数将连接时限值设置为 15 秒）。

技巧 284 使用 SQLConnect 和 SQLDisconnect 建立和结束 DBMS 会话

为了访问数据库，必须首先使用合法的具有合适权限的用户名/密码对登录到 DBMS 上，以便进行工作。例如，每当启动 MS-SQL Server Query Analyzer 的另一个实例，程序都询问输入用户名和密码。Query Analyzer 接着将此信息传递到 DBMS 加以认证。如果键入了合法的用户名/密码对，Query Analyzer 就建立与 DBMS 的连接并将 SQL 语句传递到系统加以执行。

SQLConnect 函数允许 C 程序建立与 DBMS 的连接。在建立与 SQL 服务器连接之后，应用程序可向 DBMS 发送 SQL 语句用于执行。

SQLConnect 函数调用的语句为：

```
RETCODE SQLConnect(hdbc, szDSN, cbDSN, szUID, cbUID,
szAuthStr, cbAuthStr)
```

其中：

- hdbc 是由 SQLAllocConnect 函数返回的连接句柄。
- szDSN 是以 null 结尾的字符串, 其中有与 DBMS 连接时 ODBC 驱动程序使用的数据源名称。
- cbDSN 是 szDSN 参数中 DSN 的长度。
- szUID 是以 null 结尾的字符串, 其中有登录到 DBMS 时使用的用户名。
- cbUID 是 cbDSN 参数中用户名的长度。
- szAuthStr 是用户的认证字符串(密码)。
- cbAuthStr 是 szAuthStr 参数中密码的长度。

因而, 为了以用户名 konrad 的身份, 密码为 king 登录到默认的和 DSN 为 MSSQLServer 的相关联数据库中, 应该首先将参数值(而不是 hdbc)放到适当类型的变量中, 如下所示:

```
unsigned char data_source_name[]="MSSQLServer";
unsigned char user_ID[]="konrad";
unsigned char password[]="king";
```

然后调用 SQLConnect 函数:

```
retcode = SQLConnect(hdbc, data_source_name, SQL_NTS,
                    user_ID, SQL_NTS, password, SQL_NTS);
```

注意: 在 SQL.H 中, SQL_NTS 的值(或 SQL 的以 NULL 结尾的字符串)定义为常数-3。如果将任何一个(或所有的)字符串的长度参数(cbDSN、cbUID、cbAuthStr)设置为 SQL_NTS, 系统将决定在其他参数中传递的字符串的长度。

为了结束 DBMS 上的会话, 可调用 SQLDisconnect 函数并向其传递想要结束的会话的连接句柄。例如, 在本例中, 当应用程序完成所要执行的工作时, 可让程序执行以下语句:

```
SQLDisconnect (hdbc);
```

在将 SQLConnect 与 SQLDisconnect 函数调用添加在技巧 281 “使用 SQLAllocEnv 和 SQLFreeEnv 分配及释放 ODBC 环境资源”中开始的示例程序, 以创建并中断与 DBMS 的连接之后, 应用程序就能够采取与 DBMS 连接所需的所有步骤, 然后断开并释放所有句柄和分配给 ODBC 驱动程序的内存:

```
#INCLUDE <stdio.h>
#include <windows.h>
#include <sqlext.h>
int WINAPI WinMain(HINSTANCE hInstance,
                  HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine,
                  int iCmdShow)
{
    HENV henv;                //HENV is data type: void*
    HDBC hdbc;                //HDBC is data type: void*
    RETCODE retcode;          //signed short (16-Bit) integer
    LPSTR retcode_text = " "; //character string
    unsigned char data_source_name[]="MSSQLServer";
    unsigned char user_ID[]="konrad";
    unsigned char password[]="king";
    retcode = SQLAllocEnv(&henv);
```

```

if (retcode == SQL_SUCCESS)
{
    retcode = SQLAllocConnect(henv, &hdbc);
    if (retcode == SQL_SUCCESS)
    {
        //connect to a DBMS
        retcode = SQLConnect(hdbc,data_source_name,SQL_NTS,
                             user_ID,SQL_NTS,password,SQL_NTS);
        if (retcode == SQL_SUCCESS ||
            retcode == SQL_SUCCESS_WITH_INFO)
        {
            /* SEND STATEMENTS FOR EXECUTION BY DBMS HERE(Tip 377) */

            MessageBox(0,"Connection to MSSQLServer, OK!",
                       "Connect to MSSQLServer", MB_OK);
            SQLDisconnect(hdbc); //disconnect from DBMS
        }
        else
            sprintf(retcode_text,"Error on SQLConnect = %d",
                    retcode);
        SQLFreeConnect(hdbc);
    }
    else
        sprintf(retcode_text,"Error on SQLAllocConnect = %d",
                retcode);
    SQLFreeEnv(henv);
}
else
    sprintf(retcode_text,"Error on SQLAllocEnv = %d",
            retcode);

/* display any error messages */
if (strcmp(retcode_text," ") != 0)
    MessageBox (0, retcode_text, "Connect to MSSQLServer",
                MB_OK|MB_ICONERROR);
return 0;
}

```

在本技巧中，已经完成了允许与任何定义了与有数据源名称（DSN）和有 ODBC 驱动程序 DBMS 连接的 C 程序。

技巧 285 使用 SQLAllocEnv 和 SQLFreeEnv 分配及释放 SQL 语句句柄和内存资源

在 C 程序中进行了必要的函数调用建立与 SQL 服务器的连接（正如在技巧 281~284 中所学习的）之后，可使用该连接向 DBMS 发送用于执行的 SQL 语句。但是，在调用 SQLExecDirect 向 SQL 服务器发送命令之前，必须调用 SQLAllocStmt 函数为 SQL 语句信息以及指向此区域的语句句柄保留内存区域。

SQLAllocStmt 函数调用头部的句法如下：

```
RETCODE SQLAllocStmt (hdbc, phstmt)
```

其中：

- hdbc 是连接句柄，通过这一句柄可向 DBMS 发送 SQL 语句（在技巧 282 “使用 SQLAllocConnect 和 SQLFreeConnect 分配和释放连接句柄和内存资源”中，读者学习了如何调用 SQLAllocConnect 函数来分配连接句柄）。
- phstmt 是指向系统保存语句句柄的内存地址的指针（语句句柄是一个指向程序可找到有关语句的当前处理状态、错误代码和消息、语句游标名以及所包括的列数等信息的内存位置的指针）。

因此函数返回的 RETCODE 值可以是 SQL_SUCCESS、SQL_SUCCESS_WITH_INFO、SQL_INVALID_HANDLE 或 SQL_ERROR。

因此，当应用程序使用以下语句调用 SQLAllocStmt 函数时：

```
retcode = SQLAllocStmt (hdbc, &hstmt);
```

ODBC 驱动程序为语句信息分配内存并将语句句柄（指向语句内存区域的指针）的值保存在变量 hstmt 中。

注意：在调用 SQLAllocStmt 之前，程序必须调用 SQLAllocConnect 来分配连接句柄（在本例中是 hdbc），然后使用连接句柄通过调用 SQLConnect 来建立与 DBMS 的连接。

当程序完成向 DBMS 发送语句时，可调用 SQLFreeStmt 释放语句句柄和内存资源。SQLFreeStmt 函数的头句法是：

```
SQLFreeStmt(hstmt, uiOption)
```

其中：

- hstmt 是语句句柄（用 SQLAllocStmt 函数调用来调用）。
- uiOption 是以下选项之一：
 - SQL_CLOSE——关闭与语句句柄 hstmt 相关联的游标并丢弃所有未决的结果。
 - SQL_DROP——释放 hstmt 语句句柄，关闭游标（丢弃所有未决的结果），并释放与此句柄相联系的内存资源。
 - SQL_UNBIND——通过 SQLBindCol 函数调用释放所有与 hstmt 语句句柄绑定在一起的列缓冲区。
 - SQL_RESET_PARAMS——通过 SQLBindParameter 函数调用释放所有与 hstmt 语句句柄绑定在一起的参数缓冲区。

请复核一下技巧 286 “使用 SQLExecDirect 向 DBMS 发送用于执行的 SQL 语句”中的 C 源程序。现在要理解的重要事情是，程序必须调用 SQLAllocStmt 函数分配语句句柄和内存区域，通过语句句柄 ODBC 驱动程序可将 SQL 语句传递到 DBMS。另外，仅当应用程序成功地分配了连接句柄之后，程序才能分配语句句柄（以及内存资源），（请参见技巧 281 “使用 SQLAllocEnv 和 SQLFreeEnv 分配及释放 ODBC 环境资源”）并将其用于建立与 DBMS 的连接（参见技巧 284 “使用 SQLConnect 和 SQLDisconnect 建立和结束 DBMS 会话”）。

技巧 286 使用 SQLExecDirect 向 DBMS 发送用于执行的 SQL 语句

在调用 SQLAllocStmt 函数分配了语句句柄之后，可调用 SQLExecDirect 函数向数据源（也

就是已经连接的 DBMS) 发送 SQL 语句。使用 ODBC 接口的好处是, 只需确保 SQL 语句在句法上是正确的。ODBC 驱动程序在向目标 DBMS 发送用于执行的语句之前, 将修改语句, 使之与数据源所使用的特定 SQL 形式相符合。

SQLExecDirect 函数的句法如下:

```
RETCODE SQLExecDirect(hstmt, szSQLStmt, cbSQLStmt)
```

其中:

- hstmt 是通过调用 SQLAllocStmt 函数分配 (在技巧 285 “使用 SQLAllocEnv 和 SQLFreeEnv 分配及释放 SQL 语句句柄和内存资源” 中) 的语句句柄。
- szSQLStmt 是包括想要让 DBMS 执行的 SQL 语句的字符串。
- cbSQLStmt 是 szSQLStmt 中字符串长度 (SQL 语句)。

由此函数返回的 RETCODE 类型的值可为 SQL_SUCCESS、SQL_SUCCESS_WITH_INFO、SQL_NEED_DATA、SQL_STILL_EXECUTING、SQL_ERROR 或 SQL_INVALID_HANDLE。

例如, 假设想让 DBMS 对 NORTHWIND (示例的) 数据库中的 PRODUCTS 表执行以下 UPDATE 语句:

```
UPDATE products SET unitprice = unitprice * 1.20
```

假设已经通过数据源名称指定 NORTHWIND 为默认的数据库并建立了与 DBMS 的连接, 则可调用下面的函数来执行该 UPDATE 语句:

```
void Raise_Prices(HDBC hDb_connection_handle)
{HSTMT hStatement_handle; //data type void*
RETCODE retcode;           //signed short (16-Bit Integer)
LPSTR retcode_text = " "; //character string
retcode = SQLAllocStmt(hDb_connection_handle,
                        &hStatement_handle);

if (retcode == SQL_SUCCESS)
{retcode = SQLExecDirect(hStatement_handle, (UCHAR *)
    "UPDATE products SET unitprice = unitprice "
    "* 1.20", SQL_NTS);
if (retcode != SQL_SUCCESS &&
    retcode != SQL_SUCCESS_WITH_INFO)
    sprintf(retcode_text, "Error on SQLExecDirect = %d",
        retcode);
else
    MessageBox (0, "Sucessful action",
        "Send statements to MSSQLServer in Raise_Prices",
        MB_OK);
SQLFreeStmt (hStatement_handle, SQL_DROP);}
else
    sprintf(retcode_text, "Error on SQLAllocStmt = %d",
        retcode);
if (strcmp(retcode_text, " ") != 0)
    MessageBox (0, retcode_text, "In Raise_Prices",
        MB_OK|MB_ICONERROR);}
```

注意：在运行程序之前，一定要用自己的用户名和密码代替 iConnect_SQL_Data_Source 函数中的用户名和密码。还有，必须创建名为 MSSQLServer 的 DSN 名称，正如在技巧 276 “为开放数据库互连（ODBC）连接创建数据源名称（DSN）”中所学习的那样。

技巧 287 使用 SQLFetch 函数从 SQL 数据库中提取数据行

在技巧 280~技巧 285 中，读者学习了如何编写 C 程序与 SQL 数据库连接。接着在技巧 286 “使用 SQLExecDirect 向 DBMS 发送用于执行的 SQL 语句”中学习了如何使用 SQLExecDirect 函数向 SQL 服务器发送用于执行的 SQL 语句。虽然示例只向 DBMS 发送了一条 UPDATE 语句，但发送任何语句都是平常的事——包括允许从数据库中提取可用于 C 程序中的数据 SELECT 语句。

将数据从 SQL 数据库中提取到 C 应用程序中涉及如下 4 个步骤。

1. 必须建立与 SQL 服务器的连接（正如读者在技巧 280~技巧 285 中所学习的）。
2. 必须调用 SQLExecDirect 函数将 SELECT 语句发送到 DBMS 中（读者在技巧 286 中学习的）。
3. 必须调用 SQLBindCol 函数告诉 ODBC 驱动程序当从表列中传送数据时，要使用程序中的哪个变量。
4. 必须调用 SQLFetch 或 SQLExtendedFetch 函数从 ODBC 语句缓冲区中将数据提取到为接受数据所创建的 C 变量中。

SQLFetch 函数头的句法如下：

```
RETCODE SQLFetch(hstmt)
```

其中：

- hstmt 是语句句柄。

由此函数返回的 RETCODE 类型的值可为 SQL_SUCCESS、SQL_SUCCESS_WITH_INFO、SQL_NO_DATA_FOUND、SQL_STILL_EXECUTING、SQL_ERROR 或 SQL_INVALID_HANDLE。

要理解的重要事情是，必须对想要提取到 C 变量中的每个列数据都调用此函数一次。例如，为了从 NORTHWIND 数据库的 CUSTOMERS 表列中提取 COMPANYNAME、CONTACTNAME 和 PHONE，在 Get_Cust_Info 函数中应有 3 条 SQLBindCol 和 SQLFetch 语句：

```
void Get_Cust_Info(HDBC hDb_connection_handle)
{
    #define COMPANYNAME_LEN 40
    #define CONTACTNAME_LEN 30
    #define PHONE_LEN 24
    UCHAR szCompanyName[COMPANYNAME_LEN],
          szContactName[CONTACTNAME_LEN],
          szPhoneNumber[PHONE_LEN];
    SDWORD cbCompanyName, cbContactName, cbPhoneNumber;
    HSTMT hStatement_handle; //data type void*
    RETCODE retcode;          //signed short (16-Bit Integer)
    LPSTR retcode_text = " "; //character string
    LPSTR szCustomerString = " ";

    retcode = SQLAllocStmt(hDb_connection_handle,
```

```

        &hStatement_handle);
if (retcode == SQL_SUCCESS)
{retcode = SQLExecDirect(hStatement_handle, (UCHAR *)
    "SELECT companyname, contactname, phone FROM customers"
    " WHERE customerid LIKE 'B%'"
    " ORDER BY companyname", SQL_NTS);
if (retcode != SQL_SUCCESS && retcode !=
    SQL_SUCCESS_WITH_INFO)
    sprintf(retcode_text, "Error on SQLExecDirect = %d",
        retcode);
else
{/* Bind columns CompanyName (SELECT statement column 1),
    ContactName(SELECT statement column 2), and
    Phone(SELECT statement column 3) */
    SQLBindCol(hStatement_handle, 1, SQL_C_CHAR,
        szCompanyName, COMPANYNAME_LEN, &cbCompanyName);
    SQLBindCol(hStatement_handle, 2, SQL_C_CHAR,
        szContactName, CONTACTNAME_LEN, &cbContactName);
    SQLBindCol(hStatement_handle, 3, SQL_C_CHAR,
        szPhoneNumber, PHONE_LEN, &cbPhoneNumber);
    /* Fetch and display each row of data.
        On an error, display a message and exit. */
    while (TRUE)
    {retcode = SQLFetch(hStatement_handle);

        if (retcode == SQL_ERROR || retcode ==
            SQL_SUCCESS_WITH_INFO)
            sprintf(retcode_text, "Error on SQLExecDirect = %d",
                retcode);
        if (retcode == SQL_SUCCESS || retcode ==
            SQL_SUCCESS_WITH_INFO)
        {strcpy(szCustomerString, "Company: ");
            strcat(szCustomerString, (const char *)szCompanyName);
            strcat(szCustomerString, "\nContact: ");
            strcat(szCustomerString, (const char *)szContactName);
            strcat(szCustomerString, "\nPhone: ");
            strcat(szCustomerString, (const char *)szPhoneNumber);
            MessageBox (0, (const char *)szCustomerString,
                "In Get_Cust_Info", MB_OK);}
        else break;}}

    SQLFreeStmt (hStatement_handle, SQL_DROP);}
else
    sprintf(retcode_text, "Error on SQLAllocStmt = %d",
        retcode);
if (strcmp(retcode_text, " ") != 0)

```

```
MessageBox (0,retcode_text,"In Raise_Prices",  
            MB_OK|MB_ICONERROR);}
```

技巧 288 使用 SQLExtendedFetch 函数创建可更新的游标 (Cursor)

技巧 287 “使用 SQLFetch 函数从 SQL 数据库中提取数据行”中使用了 SQLExecDirect 函数向 DBMS 发送很可能返回不止一行数据的 SELECT 语句。ODBC 驱动程序在本地硬盘上创建一个游标 (cursor, 即一个缓冲区) 保存所提取的数据值, 从而处理从查询中返回的多行数据。当向 DBMS 发送查询的应用程序调用 SQLFetch 函数时, ODBC 驱动程序将游标的指针前进到从数据库中提取的数据的下一行并将数据值发送到程序中绑定的变量。第 2 个 SQLFetch 函数调用从游标的第二行提取数据, 第 3 个调用从第 3 行提取数据, 依此类推。

如果只需要提取列数据值而不打算通过删除游标中的行或更新其内容来更新数据库, 就可使用 SQLFetch 函数。另一方面, 如果程序需要更新游标值并让那些变量反映在数据库中, 可使用 SQLExtendedFetch 函数。

SQLExtendedFetch 函数的头部句法如下:

```
RETCODE SQLExtendedFetch (hstmt, fFetchType, irow, pcrow,  
                           rgfRowStatus
```

其中:

- hstmt 是语句句柄。
- fFetchType 是提取的类型, 取值如下: SQL_FETCH_NEXT、SQL_FETCH_FIRST、SQL_FETCH_LAST、SQL_FETCH_PRIOR、SQL_FETCH_ABSOLUTE、SQL_FETCH_RELATIVE 或 SQL_FETCH_BOOKMARK。
- irow 是从 DBMS 中提取的行数。
- pcrow 是从 DBMS 中提取的实际行数。
- rgfRowStatus 是自从从数据源 (DBMS) 中最后一次提取数据以来与行的状态变化有关的状态值矢量。可能的值是 SQL_ROW_SUCCESS (意义为行未发生变化)、SQL_ROW_UPDATED、SQL_ROW_DELETED、SQL_ROW_ADDED、SQL_ROW_ERROR 或 SQL_ROW_NOWROW (用于 irow 超过了实际提取的行数 [pcrow] 的每行)。

因此函数返回的 RETCODE 类型的值可为 SQL_SUCCESS、SQL_SUCCESS_WITH_INFO、SQL_NO_DATA_FOUND、SQL_STILL_EXECUTING、SQL_ERROR 或 SQL_INVALID_HANDLE。

注意: SQLExtendedFetch 函数从游标中提取数据是以行集为基础的, 而不是一行一行提取。行集是由主调的 SQLSetStmtOption 函数定义的一组行并设置了 SQL_ROWSET_SIZE 选项。例如, 如果调用了 SQLSetStmtOption 函数并将行集容量设置为 10, 那么每个行集由 10 个游标行组成。作为对照, 每个 SQLExtendedFetch 调用将从游标的 10 行数据中提取一列放入绑定的程序变量中。

与 SQLFetch 函数调用不同, 此函数一次只在游标中向前移动一行, 而 SQLExtendedFetch 函数允许指定想要在游标中移动的方向 (既可向前也或向后) 以及一次想要移动的行数。

例如, 假如类型定义为:

```

RETCODE retcode //signed short (defined in SQLTypes.h)
HSTMT hstmt //void* (defined in SQLTypes.h)
UDWORD pcrow
UWORD rgfRowStatus

```

下面的 `SQLExtendedFetch` 函数调用将从游标中的最新的行集中提取列值：

```

retcode = SQLExtendedFetch(hstmt, SQL_FETCH_LAST, 1,
                           &pcrow, rgfRowStatus)

```

另外，下面的调用将从第一个行集中提取列值：

```

retcode = SQLExtendedFetch(hstmt, SQL_FETCH_FIRST, 1,
                           &pcrow, rgfRowStatus)

```

同时，以下函数调用将从游标中包括第 10 行的行集中提取数据：

```

retcode = SQLExtendedFetch(hstmt, SQL_FETCH_ABSOLUTE, 10,
                           &pcrow, rgfRowStatus)

```

而如下的函数调用将从包括游标中当前行集第一行前的 4 行的行集中提取值：

```

retcode = SQLExtendedFetch(hstmt, SQL_FETCH_RELATIVE, -4,
                           &pcrow, rgfRowStatus)

```

在技巧 291~技巧 293 中，读者将学习如何在 `SQLExtendedFetch` 函数调用之后使用 `SQLSetPos` 函数调用在由 `SQLExtendedFetch` 函数生成的游标中更新、删除或是添加数据行。

技巧 289 理解行式和列式绑定之间的差别

在从 SQL 数据库中提取数据值用在以 C 这样编程语句编写的应用程序中时，ODBC 驱动程序首先取出查询生成的数据并将其保存在硬盘上的临时文件中。正如读者在技巧 288 “使用 `SQLExtendedFetch` 函数创建可更新的游标（Cursor）”中所学习的，这个临时存储区域（或缓冲区）叫做游标（cursor）。为了在程序中使用数据库数据，必须将数据从游标提取到应用程序已经声明的变量中。告诉 ODBC 驱动程序程序中的哪个变量将从游标的哪一列接受数据的过程就称为绑定。

在列式绑定中，要调用 `SQLBindCol` 函数来建立在程序中声明的变量与游标中的列之间的链接。在行式绑定中，要调用的是同一个 `SQLBindCol` 函数，但却使用此函数建立结构中的成员（或域）与游标中的列之间的链接。

如果打算使用列式绑定，就不再要求调用 `SQLStmtOption` 函数，除非以前 `SQLStmtOption` 函数来请求过行式绑定。因此，由于已经建立了数据库连接而且已经分配了语句句柄（在本例中是 `hStatementHandle`），应用程序可执行下面的代码来建立列式绑定：

```

#define TEN_ROWS 10
#define CMPNY_LEN 40
#define CONTACTNAME_LEN 30
#define PHONE_LEN 24
UCHAR szCmpnyName[CMPNY_LEN],
      szContactName[CONTACTNAME_LEN],
      szPhoneNumber[PHONE_LEN];
SDWORD cbCmpnyName, cbContactName, cbPhoneNumber;

SQLBindCol(hStatementHandle, 1, SQL_C_CHAR,

```

```

        szCmpnyName, CMPNY_LEN, &cbCmpnyName);
SQLBindCol(hStatementHandle, 2, SQL_C_CHAR,
        szContactName, CONTACTNAME_LEN, &cbContactName);
SQLBindCol(hStatementHandle, 3, SQL_C_CHAR,
        szPhoneNumber, PHONE_LEN, &cbPhoneNumber);

```

在执行了 3 个 SQLBindCol 函数调用之后,任何以后对 SQLFetch 或 SQLExtendedFetch 的调用都会使 ODBC 驱动程序将游标中的第一列中的数据值复制到变量 CmpnyName 中并在变量 cbCmpnyName 中记下保存数据所用的字节数。ODBC 驱动程序还把第二个游标列中的数据复制到 ContactName 变量中并在 cbContactName 变量中记下复制的字节数。最后,ODBC 驱动程序将游标的第 3 列数据复制到 PhoneNumber 变量中,并在 cbPhoneNumber 变量中记下复制的字节数。

由于列式绑定是默认形式,如果想要既可使用 SQLFetch 又可使用 SQLExtendedFetch 函数调用把游标数据传送到结构的成员中而不是传送到单个的变量中,则必须调用 SQLStmtOption 函数并请求行式绑定。再一次地,假设已经建立起与数据库的连接并已经分配了语句句柄(与在最后的示例中的一样是 hStatementHandle),应用程序可执行下面的代码来建立行式绑定:

```

#define CMPNYNAME_LEN 40
#define CONTACTNAME_LEN 30
#define PHONE_LEN 24

typedef struct{UCHAR szCmpnyName[CMPNYNAME_LEN],
        SDWORD cbCmpnyName;
        UCHAR szContactName[CONTACTNAME_LEN],
        SDWORD cbContactName;
        UCHAR szPhoneNumber[PHONE_LEN];
        SDWORD cbPhoneNumber;}
        CustInfoTable;
CustInfoTable citCustInfo;

SQLSetStmtOption(hStatementHandle, SQL_BIND_TYPE,
        sizeof(CustInfoTable));

SQLBindCol(hStatement_handle, 1, SQL_C_CHAR,
        citCustInfo.szCmpnyName, CMPNYNAME_LEN,
        &citCustInfo.cbCmpnyName);
SQLBindCol(hStatement_handle, 2, SQL_C_CHAR,
        citCustInfo.szContactName, CONTACTNAME_LEN,
        citCustInfo.&cbContactName);
SQLBindCol(hStatement_handle, 3, SQL_C_CHAR,
        citCustInfo.szPhoneNumber, PHONE_LEN,
        citCustInfo.&cbPhoneNumber);

```

在执行了 3 个 SQLBindCol 函数调用之后,以下对 SQLFetch 或 SQLExtendedFetch 的调用都会使 ODBC 驱动程序将游标中的第一列中的数据值复制到结构成员 CmpnyName 中,并在结构成员 cbCmpnyName 中记下保存数据所用的字节数。ODBC 驱动程序还把第二个游标列中

的数据复制到 ContactName 结构成员中并在 cbContactName 结构成员中记下复制的字节数。最后，ODBC 驱动程序将游标的第 3 列数据复制到 PhoneNumber 结构成员中，并在 cbPhoneNumber 结构成员中记下复制的字节数。

为了从行式绑定切换回列式绑定，可用定义的常数 SQL_BIND_BY_COLUMN 作为第 3 个参数来调用 SQLSetStmtOption，如下所示：

```
SQLSetStmtOption(hStatementHandle, SQL_BIND_TYPE,  
                  SQL_BIND_BY_COLUMN);
```

而不是将保存数据的结构尺寸作为第 3 个参数传递：

```
SQLSetStmtOption(hStatementHandle, SQL_BIND_TYPE,  
                  sizeof(name_of_structure));
```

技巧 290 使用 SQLSetConnectOption 函数选择在执行 SQL 语句时使用的数据库

在技巧 286 “使用 SQLExecDirect 向 DBMS 发送用于执行的 SQL 语句”中，读者学习了如何调用 SQLExecDirect 函数向数据源（DBMS）发送用于执行的 SQL 语句。接着，在技巧 287~技巧 289 中，学习了如何使用 SQLFetch 和 SQLExtendedFetch 函数从 SQL 数据库将数据提取到 C 应用程序的变量中。技巧 286~技巧 289 中的示例假设程序需要数据位于在创建数据源名称（DNS）时选定的默认数据库中（读者在技巧 276 “为开放数据库互连（ODBC）连接创建数据源名称（DSN）”中学习了如何为 SQL 服务器创建 DSN 的内容）。为了找出特定的连接句柄的数据库名称，可调用 SQLGetConnectOption 函数。

SQLGetConnectOption 函数头的句法如下：

```
RETCODE SQLGetConnectOption(hdbc, fOption, vParam)
```

其中：

- HDBC hdbc 是用于将 C 程序与 SQL 服务器连接的连接句柄。
- UWORD fOption 是想要提取其值的连接选项（为了得到选项值的清单，请参见技巧 283 “使用 SQLSetConnectOption 为与 SQL Server 的 ODBC 连接设置会话选项”中的表 15.1。）
- UDWORD vParam 既可是 32 位整数值也可是一个指向 NULL 结尾的字符串的指针，这要由函数所提取的连接选项的类型决定（由 fOption 值所确定的）。

因而，如果程序以前建立了与 SQL 服务器的连接并声明了变量：

```
HDBC hdbc;  
LPSTR szDatabaseToUse = " ";  
RETCODE retcode;
```

以下函数调用将把语句通过连接句柄 hDb_connection_handle 向 DBMS 发送的数据库名放入变量 szDatabaseToUse 中：

```
retcode = SQLGetConnectOption(hDb_connection_handle,  
                               SQL_CURRENT_QUALIFIER, szDatabaseToUse);
```

为了改变连接句柄所用的数据库，可调用 SQLSetConnectOption 函数。SQLSetConnectOption 函数与 SQLGetConnectOption 函数头有相同的句法。但是，在 SQLSetConnectOption 函数调用中，vParam 参数是输入参数。因而为了让 DBMS 在执行通过连接句柄 hDb_connection_handle 发送的语句时使用 SQLTips 数据库，可执行如下

SQLSetConnectOption 函数调用:

```
retcode = SQLSetConnectOption(hDb_connection_handle,  
    SQL_CURRENT_QUALIFIER, (UDWORD) "SQLTips");
```

技巧 291 使用 SQLSetPos 函数设置行集中的游标位置

SQLSetPos 是一个多目的的函数, 不仅允许设置查询结果的游标中指向行集中的特定行的指针, 而且还允许刷新并 (或) 改变游标的内容。请记住, 当调用 SQLSetPos 函数修改游标内容 (通过在函数调用中指定 SQL_UPDATE、SQL_DELETE 或 SQL_ADD) 时, ODBC 驱动程序将把 SQL 语句发送到 DBMS 中, 以便对游标的位于 SQL 服务器上的底层数据做出改变。

SQLSetPos 函数头的句法如下:

```
RETCODE SQLSetPos(hstmt, irow, fOption, fLock)
```

其中:

- HSTMT hstmt 是语句句柄。
- UWORD irow 是对其执行由 fOption 参数指定的操作的行集的行数。
- UWORD fOption 是在游标中以及语句句柄通过其连接句柄连接其上的数据源上的数据库中执行的操作。可能的 fOption 操作是 SQL_POSITION、SQL_REFRESH、SQL_UPDATE、SQL_DELETE 和 SQL_ADD。
- fLock 用于通过对不支持那些事务处理的数据源的事务处理的模拟控制数据库并发性控制。虽然 fLock 可设置为 SQL_LOCK_NO_CHANGE、SQL_LOCK_EXCLUSIVE、或 SQL_LOCK_UNLOCK, 但只有 SQL_LOCK_NO_CHANGE 选项是被 SQL 服务器游标所支持的, 因为 DBMS 有其自己的隔离级别 (并发性) 和事务处理支持。

由函数返回的 RETCODE 类型的值可为 SQL_SUCCESS、SQL_SUCCESS_WITH_INFO、SQL_NEED_DATA、SQL_STILL_EXECUTING、SQL_ERROR 或 SQL_INVALID_HANDLE。

例如, 为了将游标的指针定位于行集的第 5 行上, 可执行以下语句:

```
retcode = SQLSetPos(hstmt, 5, SQL_POSITION,  
    SQL_LOCK_NO_CHANGE)
```

在定位游标指针之后 (常常称为 “定位游标”), SQL_POSITION 选项告诉 ODBC 驱动程序不要采取进一步的行动。正如读者将在技巧 292 和技巧 293 中所学习的, 可通过在 SQLSetPos 函数调用以 SQL_UPDATE 或 SQL_DELETE 代替 SQL_POSITION 让 ODBC 驱动程序更新或删除数据库表中的一行。

技巧 292 使用 SQLSetPos 函数的 SQL_UPDATE 选项执行定位更新

如果数据源的 ODBC 驱动程序允许的话, 程序可使用 SQLSetPos 函数的 SQL_UPDATE 选项来修改游标中的数据值并使改变反映在游标从中提取数据行的数据库中。当应用程序用 SQL_UPDATE 选项调用 SQLSetPos 函数时, ODBC 驱动程序将游标指针定位于作为 irow 参数的值传递到函数的行集的行上。接着, ODBC 驱动程序通过将数据值从游标行缓冲区中发送到游标从中提取数据的表行的列中, 从而更新数据库中的底层行数据。

例如, 假设程序调用 SQLSetStmtOption 函数将其行集尺寸设置为 10, 然后调用 SQLExecDirect 函数 (参见技巧 286 “使用 SQLExecDirect 向 DBMS 发送用于执行的 SQL 语句”) 将以下 SQL 语句发送到 DBMS 用于执行:

```
SELECT cust_ID, f_name, l_name, phone_number
FROM customers384
```

在调用 `SQLBindCol` 函数将游标中的缓冲区绑定到程序中的变量（正如读者在技巧 287~技巧 289 中所学习的）之后，应用程序可根据通过以下代码提取的第 12 行中的数据更新绑定到 `szFirstName` 和 `szLastName` 的表列：

```
retcode = SQLExtendedFetch(hStatementHandle,
    SQL_FETCH_ABSOLUTE, 12, &pcrow, rgfRowStatus);

if (rgfRowStatus[0] != SQL_ROW_DELETED &&
    rgfRowStatus[0] != SQL_ROW_ERROR)
{
    strcpy(citCustInfoRecord[0].szFirstName, "Konrad");
    strcpy(citCustInfoRecord[0].szLastName, "King");
    SQLSetPos(hStatementHandle, 1, SQL_UPDATE,
        SQL_LOCK_NO_CHANGE)
}
```

在本例中，`SQLExtendedFetch` 函数调用从游标中从第 12 行算起的第二个行集中提取行。由于 `citCustInfoRecord` 结构的数组已经通过 `SQLBindCol` 函数调用（没有显示出来）绑定到游标缓冲区上，数组中的第一个元素（索引值为 0）在 `SQLExtendedFetch` 函数调用之后包括来自游标中第 12 行的值。接着，`strcpy` 函数调用通过改变结构数组的第一行中的结构元素 `szFirstName` 和 `szLastName` 的值更新当前行集（游标中的第 12 行）中的游标缓冲区。最后，`SQLSetPos` 函数调用告诉 ODBC 驱动程序将当前行集第一行（游标中的第 12 行）缓冲区中的值复制到 `SQLExtendedFetch` 函数调用原来提取表行的底层表列中。

注意：许多 ODBC 驱动程序不支持使用 `SQLSetPos` 函数的 `SQL_UPDATE` 选项的定位更新。如果正在使用的数据库的 ODBC 驱动程序就是不支持上述更新，应用程序仍然可以更新数据库中的列值。但是，必须使用 `SQLExecDirect` 一类的函数而不是 `SQLSetPos` 的 `SQL_UPDATE` 选项。技巧 295 “在 ODBC 驱动程序不支持定位更新时使用 `SQLExecDirect` 函数更新数据库中的列值”将向读者展示，在游标已经打开的情况下，如何使用 `SQLExecDirect` 函数向 DBMS 发送 `UPDATE` 语句。

技巧 293 使用 `SQLSetPos` 函数的 `SQL_DELETE` 选项执行定位删除

正如读者在技巧 292 “使用 `SQLSetPos` 函数的 `SQL_UPDATE` 选项执行定位更新”中所学习的，在使用 `SQLSetPos` 函数更新游标缓冲区中的值时，ODBC 驱动程序向 DBMS 发送命令，对派生出游标的底层数据库中的表行做出同样的改变。类似地，在使用 `SQLSetPos` 函数的 `SQL_DELETE` 选项时，ODBC 驱动程序将光标定位于由 `irow` 参数指定的行上，告诉 DBMS 删除数据库表中的底层行并将游标行的状态标志（在 `rgfRowStatus` 数组中）改变为 `SQL_ROW_DELETED`。

正如读者在技巧 291 “使用 `SQLSetPos` 函数设置行集中的游标位置”中所学习的，`SQLSetPos` 函数头的句法如下：

```
RETCODE SQLSetPos(hstmt, irow, fOption, fLock)
```

因而，为了删除游标中的第 9 行中的底层表中的行，应用程序应执行以下代码：

```

retcode = SQLExtendedFetch(hStatementHandle,
    SQL_FETCH_ABSOLUTE, 9, &pcrow, rgfRowStatus);
if (rgfRowStatus[0] != SQL_ROW_DELETED &&
    rgfRowStatus[0] != SQL_ROW_ERROR)
    SQLSetPos(hStatementHandle, 1, SQL_DELETE,
        SQL_LOCK_NO_CHANGE)

```

请注意, SQLSetPos 函数调用中 irow 参数的值已被设置为 1 而不是 9。正如读者在技巧 288“使用 SQLExtendedFetch 函数创建可更新的游标(Cursor)”中所学习的, SQLExtendedFetch 函数一次从游标中提取一个行集的数据。由 SQLExtendedFetch 函数调用提取的行集中的第一行是作为 irow 参数传递的行——在本例中是 9。因而, 为了处理游标中的第 9 行, 应用程序必须告诉 ODBC 驱动程序使用行集中的第 1 行。

在使用 SQLSetPos 的 SQL_DELETE 选项时, 一定要将 irow 参数的值设置为不等于 0 的值, 除非想让 ODBC 驱动程序删除行集中所有行对应的底层表行。例如, 虽然下面的函数调用将删除游标行集第 2 行对应的底层表行:

```
SQLSetPos(hstmt, 2, SQL_DELETE, SQL_LOCK_NO_CHANGE)
```

但下面的函数调用却删除行集中的所有行:

```
SQLSetPos(hstmt, 0, SQL_DELETE, SQL_LOCK_NO_CHANGE)
```

因而, 如果行集包括 10 行, 第一个 SQLSetPos 函数调用将删除单行, 虽然第二个 SQLSetPos 函数调用删除 10 行。

注意: 与 SQL_UPDATE 选项的情况一样, 某些 ODBC 驱动程序不支持使用 SQLSetPos 函数的 SQL_DELETE 选项的定位删除。如果正在使用的数据库源的 ODBC 驱动程序不支持, 程序仍然可以通过调用 SQLExecDirect 函数(读者将在技巧 294“当 ODBC 驱动程序不支持定位删除时使用 SQLExecDirect 函数删除数据库中的行”中学习有关内容)来删除数据库中的行。

技巧 294 当 ODBC 驱动程序不支持定位删除时使用 SQLExecDirect 函数删除数据库中的行

下面的 SQL DELETE 语句称为搜索的 (searched) DELETE 语句:

```
DELETE FROM customers WHERE cust_id = 9
```

因为 DBMS 要搜索满足 WHERE 子句中的搜索条件的目标表行, 然后将其删除。另一方面, 术语“定位的” (positioned) 删除只适用于那些删除由游标中当前行所引用的数据库表中的单个底层行的编程性 SQL 语句。类似地, 如下的 SQL UPDATE 语句是搜索的 (searched) UPDATE 语句:

```
UPDATE customers SET f_name = 'Konrad', i_name = 'king'
WHERE cust_id = 9
```

因为 DBMS 在扫描 CUSTOMERS 表中的行时, 使用 WHERE 子句中的搜索条件来决定更新哪些行。与定位的 DELETE 语句类似, 一条定位的 UPDATE 语句改变由游标中当前行所引用的数据库表中的单个底层行的列中的数据值。

如果 ODBC 驱动程序不支持使用 SQLSetPos 函数, 通过调用 SQLExecDirect 函数既可模拟定位更新也可模拟定位删除, 此函数带有 UPDATE 或 DELETE 语句字符串, 其中的 WHERE 子句将“当前的”游标行与其数据库表中的底层行链接起来。例如, 如果底层表包括名为 TSTAMP 的类型为 TIMESTAMP 的列, 程序可通过如下代码对游标中的第 5 行的数据库表的

底层行执行“定位的”删除：

```
strcpy(szSQLStatement,  
      "DELETE FROM customers WHERE tstamp = 0x");  
strcat(szStatementString, citCustInfoRecord[0].szTimeStamp;  
retcode = SQLExecDirect(hDeleteStmtHandle,  
      (UCHAR *)szSQLStatement, SQL_NTS)
```

请注意，虽然在本例中由 SQLExecDirect 函数发送到 DBMS 的 DELETE 语句的“作用好像是”定位的删除，因为它告诉 DBMS 删除游标从中提取数据的表中的一行，但实际上是搜索的 DELETE 语句。由于类型为 TIMESTAMP 的列的每行中的值保证在整个数据库中是惟一的，在本例中的搜索的 DELETE 语句总是从底层表中删除至少一行（如果另一用户已经删除了游标的底层行的话，删除 0 行也是可能的）。

如果底层表中没有类型为 TIMESTAMP 的列，仍然可以模拟定位删除。只要将 WHERE 子句中的搜索条件改变为在任何列约束为 UNIQUE 的列中查找底层表的 PRIMARY KEY 列之一的值，或者在某些其他复合值对底层表中的每行都是惟一的列组合中查找值。

注意：如果想要在游标已经打开的情况下调用 SQLExecDirect 函数，必须首先分配第二个语句句柄。否则 SQLExecDirect 函数调用将会指明“Invalid Cursor State.”（非法的游标状态。）错误代码中止。

技巧 295 在 ODBC 驱动程序不支持定位更新时使用 SQLExecDirect 函数更新数据库中的列值

在技巧 292“使用 SQLSetPos 函数的 SQL_UPDATE 选项执行定位更新”中，读者了解到，可使用 SQLSetPos 的 SQL_UPDATE 选项对游标的数据库的底层行执行定位更新。在执行定位更新时，ODBC 驱动程序告诉 DBMS 更新与游标缓冲区相匹配的派生出游标的当前行的数据库中的底层表行中的值。简短地说，定位更新只不过是一种搜索更新，其中 WHERE 子句中的搜索条件惟一地标识出与游标中的当前行对应的底层行。

遗憾的是，许多 ODBC 驱动程序不支持定位更新（或更新删除）。结果，必须使用 SQLExecDirect 函数来模拟定位更新，正如在技巧 293“使用 SQLSetPos 函数的 SQL_DELETE 选项执行定位删除”中使用该函数模拟定位删除一样。例如，假设有一个游标，其中的行是由用以下查询调用 SQLExecDirect 函数产生的：

```
retcode = SQLExecDirect(hQueryStmtHandle, (UCHAR *)  
      "SELECT tstamp, cust_ID, f_name, l_name, phone_number"  
      " FROM customers", SQL_NTS);
```

在调用了将游标缓冲区与 C 程序中的变量绑定的 SQLBindCol 函数之后，可执行如下语句来更新游标中的第 5 行：

```
retcode = SQLExtendedFetch(hQueryStmtHandle, SQL_FETCH_ABSOLUTE, 5,  
      &pcrow, &gfrRowStatus);  
strcpy(citCustInfoRecord[0].szFirstName, "Sally");  
strcpy(citCustInfoRecord[0].szLastName, "Wells");
```

接着，执行以下语句让 DBMS 对 CUSTOMERS 表中的游标的底层行做出同样的改变：

```
strcpy(szStmtString, "UPDATE customers SET f_name = '");  
strcat(szStmtString, citCustInfoRecord[0].szFirstName);
```

```

strcat(szStmtString, " ", l_name = "
strcat(szStmtString, citCustInfoRecord[0].szLastName);
strcat(szStmtString, " WHERE tstamp = 0x");
strcat(szStmtString, citCustInfoRecord[0].szTimeStamp);

```

```

retcode = SQLExecDirect(hUpdateStmtHandle,
                        (UCHAR *)szStmtString, SQLNLS);

```

虽然技巧 293 和本技巧中的示例中的 WHERE 子句的搜索条件都使用了 TIMESTAMP 类型的列（列名 TSTAMP）来惟一地标识游标行的数据库底层行，但任何受 UNIQUE 约束的列都可起同样的作用。例如，如果本例中的 CUST_ID 列对游标的底层表中的每一行都是惟一的，那么执行以下语句将与前例中使用在调用 SQLExecDirect 函数时使用 TSTAMP 列值作为搜索条件的效果是一样的：

```

strcpy(szStmtString, "UPDATE customers SET f_name = ");
strcat(szStmtString, citCustInfoRecord[0].szFirstName);
strcat(szStmtString, " ", l_name = "
strcat(szStmtString, citCustInfoRecord[0].szLastName);
strcat(szStmtString, " WHERE cust_ID = ");
strcat(szStmtString, citCustInfoRecord[0].szCust_ID);

```

```

retcode = SQLExecDirect(hUpdateStmtHandle,
                        (UCHAR *)szStmtString, SQLNLS);

```

注意：如果想要在将 SELECT 语句发送到 DBMS 执行从而打开游标之后，调用 SQLExecDirect 函数将 UPDATE 语句发送到 DBMS，则必须首先分配第二个语句句柄。否则，SQLExecDirect 函数调用将以指明“Invalid Cursor State.”（非法的游标状态。）的错误代码中止执行。

技巧 296 使用 SQLError 函数提取并显示 ODBC 错误代码和错误消息

当调用一个 ODBC 函数时，ODBC 驱动程序使用一个或多个通讯句柄与数据源通信。例如 SQLConnect 函数创建并使用连接句柄（数据类型为 HDBC）。同时，SQLAllocEnvironment 函数使用数据库连接句柄中的值并创建环境句柄（数据类型为 HENV）。最后 SQLAllocStmt 函数使用连接句柄中的值并创建语句句柄（数据类型为 HSTMT）。任何 ODBC 函数调用可在所创建或使用的句柄内发布 0 个或多个错误、警告或是通知消息。另外，只要函数返回了返回代码（RETCODE 为）SQL_ERROR 或 SQL_SUCCESS_WITH_INFO，必定在函数所使用的通讯句柄内至少发现一条消息。SQLError 函数允许从其最右边的非 NULL 句柄参数的数据结构中提取消息，以及从每个句柄可保存的多达 64 条的消息堆栈中删除消息。

SQLError 函数头的句法如下：

```

RETCODE SQLError(henv, hdbc, hstmt, szSQLState,
                 pfNativeError, szErrorMessage, cbErrorMsgMax,
                 pcbErrorMsg);

```

其中：

- HENV henv 是环境句柄或 SQL_NULL_HENV。
- HDBC hdbc 是一个数据库句柄或 SQL_NULL_HDBC。

- HSTMT hstmt 是一个语句句柄 SQL_NULL_HSTMT。
- UCHAR SQLState 是 ODBC 驱动程序映射的数值代码强制转换的以 NULL 结尾的字符串。
- SDWORD FAR* pfNativeError 是指向 32 位数值变量（内存位置）的指针，其中保存有数据固有的（与数据源有关的）错误代码，而此错误代码又要按 ANSI SQL 规范中指定的映射到 ODBC 驱动程序错误代码。
- UCHAR FAR* szErrorMessage 是指向保存有以 NULL 结尾的字符串的内存位置的指针，其中保存着错误信息。
- SWORD cbErrorMsgMax 是错误信息存储区域 szErrorMessage 的最大长度。其值必须小于或等于 SQL_MAX_MESSAGE_LENGTH-1（或 511，在本书写作时）。
- SWORD FAR* pcbErrorMsg 是指向 16 位数值变量（内存位置）的指针，该内存位置处保存着存放于错误消息区域 szErrorMessage 中的非 NULL 字节数。

由此函数返回的 RETCODE 类型的值可为 SQL_SUCCESS、SQL_SUCCESS_WITH_INFO、SQL_NO_DATA_FOUND、SQL_ERROR 或 SQL_INVALID_HANDLE。

因此，为了提取环境句柄中的错误消息，可用合法的环境句柄（henv）以及设置为 NULL 的 hdbc 和 hstmt 参数调用 SQLError 函数，如下所示：

```
retcode = SQLError(henv, SQL_NULL_HDBC, SQL_NULL_HSTMT,
    szSQLState, &pfNativeError, szErrorMessage,
    MSG_BUFF_SIZE, &cbErrorMsg);
```

类似地，为了提取与连接句柄有关的错误，可用合法的连接句柄（hdbc）和 NULL 的 henv 和 hstmt 参数调用 SQLError 函数，如下所示：

```
retcode = SQLError(SQL_NULL_HENV, henv, SQL_NULL_HSTMT,
    szSQLState, &pfNativeError, szErrorMessage,
    MSG_BUFF_SIZE, &cbErrorMsg);
```

最后，为了提取在语句句柄中发布的错误，可使用合法的语句句柄（hdbc）以及设置为 NULL 的 henv 和 hdbc 参数调用 SQLError 函数，如下所示：

```
retcode = SQLError(SQL_NULL_HENV, SQL_NULL_HENV, hstmt,
    szSQLState, &pfNativeError, szErrorMessage,
    MSG_BUFF_SIZE, &cbErrorMsg);
```

正如以前提到过的，每个 SQLError 函数调用从最右边的非 NULL 句柄中提取一种错误代码（和消息）。为了显示在 3 种句柄结构中的任何一个中发布的多条错误，可多次调用 SQLError 函数。如果调用了 SQLError 函数，而在句柄结构中没有（多余）错误消息，SQLError 将返回 retcode 类型的值为 SQL_NO_DATA_FOUND，而 szSQLState 值将为 00000，pfNativeError 将为未定义的，而 szErrorMsg 中将只包括单个的以 NULL 结尾的字节。

技巧 297 在宿主程序变量中处理 NULL 值

大多数编程语句不支持 SQL 的 NULL 值。例如，在 C 中，一个声明为

```
SWORD sSalespersonID
```

的变量可存放 16 位整数值。但是，虽然 sSalespersonID 的值可为负数、0 或正数，但它不能有未知的或缺失的值。简短地说，sSalespersonID 的值不能为 NULL。因此，当从可能包括一个

或多个 NULL 值的 SQL 表列中提取数据时，必须检查作为 SQLBindCol 函数的 pcbValue 参数而传递并提取到变量中的值。

例如，假如 SQLBindCol 的句法为：

```
SQLBindCol(hstmt, icol, fCType, rgbvalue, cbValueMax,  
           pcbValue)
```

以下程序语句将把 ODBC 游标中的第 4 个缓冲列绑定到变量 sSalespersonID 上：

```
SQLBindCol(hStatementHandle, 4, SQL_C_SSHORT,  
           &sSalespersonID, 0, &cbSalespersonID);
```

每当应用程序调用 SQLFetch（参见技巧 287 “使用 SQLFetch 函数从 SQL 数据库中提取数据行”）或 SQLExtendedFetch（参见技巧 288 “使用 SQLExtendedFetch 函数创建可更新的游标（Cursor）”）函数时，ODBC 驱动程序将把游标中的第 4 列中的值放入变量 sSalespersonID 中并将传送的字节数放在变量 cbSalespersonID 中——除非第 4 列中的值为 NULL。如果游标列的值为 NULL，而程序又调用了 SQLFetch 或 SQLExtendedFetch 函数，ODBC 驱动程序就不改变与游标缓冲区绑定的程序变量中的值。但是，ODBC 驱动程序确实将 pcbvalue 参数的值（在本例中是 cbSalespersonID）设置为 SQL_NULL_DATA，其值为-1。

因而，在使用有可能包括 NULL 数据值的 SQL 列中的值之前，一定要让程序检查从游标提取的字节数的值（pcbValue 值）。例如，下面的 if 语句：

```
if (cbSalespersonID != SQL_NULL_DATA)  
    sprintf((char *)szSalespersonID,  
           "\nSalesperson ID: %d", sSalespersonID);  
else  
    strcpy((char *)szSalespersonID,  
           "\nSalesperson ID: **UNASSIGNED **");
```

将把游标中第 4 列的销售人员 ID 值放入 szSalespersonID 字符串，如果该列有非 NULL 值的话。否则该字符串将把销售人员 ID 显示为** UNASSIGNED **（未赋值）。

技巧 298 向 Visual Basic (VB) 中添加 DB 函数库 (DBLIB) 功能

技巧 280~技巧 297 向读者展示了如何使用开放数据库连接 (ODBC) 驱动程序来处理 SQL 服务器上的数据。使用在技巧 276 “为开放数据库互连 (ODBC) 连接创建数据源名称 (DSN)”中创建的数据源名称 (DSN)，ODBC 应用程序接口 (API) 允许使用各种不同的数据源，其中可能有 MS-SQL Server。另一方面，DB 函数库 (DBLIB) 是 Microsoft 提供的 MS-SQL Server 固有的数据访问技术，只可用于访问 MS-SQL Server 上的数据。

Visual Basic (VB) 上使用的 DBLIB 是 C 上使用的 DBLIB 的子集。两者都包括在 MS-SQL Server 中。但是，不管 C 还是 VB 的 DBLIB 都不能作为默认的 MS-SQL Server 客户安装过程的一部分来安装。为了安装 DBLIB，应运行 MS-SQL Server 的定制 (Custom) 安装并选择 Development Tools (开发工具) 选项。在硬盘上安装了 VB DBLIB 之后，就可在 VB 工程中使用其中的函数，从而发出处理 MS-SQL Server 数据的 API 调用，其过程与使用 ODBC API 时差不多。

由于启用了 DBLIB 的 VB 应用程序可调用许多在 VBSQL.OCX 动态链接库 (DLL) 中定义的 DBLIB 函数，因而必须将 VBSQL.OCX 添加到 VB 工具箱中。为达此目的，可执行以下

步骤：

1. 如果还没有做这件事，请启动 MS-Visual Basic (VB) 并打开一个 Standard .EXE 工程。

2. 右击工程的工具箱（在 VB 应用程序窗口的左边）并从弹出的菜单中选择 Components（组件）命令。VB 将显示 Components 对话框（与技巧 278 “向 Visual Basic (VB) 窗体中添加 MSFlexGrid 控件以显示 SQL 表数据” 中的图 15.7 类似）。

3. 使用 Controls（控件）选项卡上 Components 列表框右边的滚动条，滚动控件组件列表，直到看到 Vbsql 控件。接着单击 Vbsql OLE Custom Control Module 左边的复选框使之出现选择标记。（控件是以字母顺序列出的，因而所要的控件将接近列表的底部，因为其是以 V 开头的。）

注意：VB 将试图在安装 MS-SQL Server 的开发工具的文件夹中查找 Vbsql 控件。对于 MS-SQL Server 7.0 来说，默认的文件夹是 C:\MSSQL7\DevTools\Lib\。如果将 VBSQL.OCX 文件移动到其他文件夹中，则需单击 Browse（浏览）按钮并搜索，或者键入 VBSQL.OCX 文件所在文件夹的全路径。

4. 单击 OK 按钮。VB 将返回 VB 应用程序窗口并将 Vbsql 控件作为最新的控件组件添加到 VB 的工具箱中。

在使用 Vbsql 控件中的函数之前，必须在 .BAS 或 .CLS 模块中声明其函数头（即其名称和参数）。幸运的是，Microsoft 为 Vbsql 控件提供了 VBSQL.BAS 模块，可将其添加到 VB 环境中，为达到此目的，请执行以下附加步骤：

1. 右击 VB 应用程序窗口右上边的 Project（工程）窗口。接着将鼠标指针移动到弹出菜单的 Add 选项上并选择 Module（模块）选项。VB 将显示 Add Module（添加模块）对话框，如图 15.12 所示。

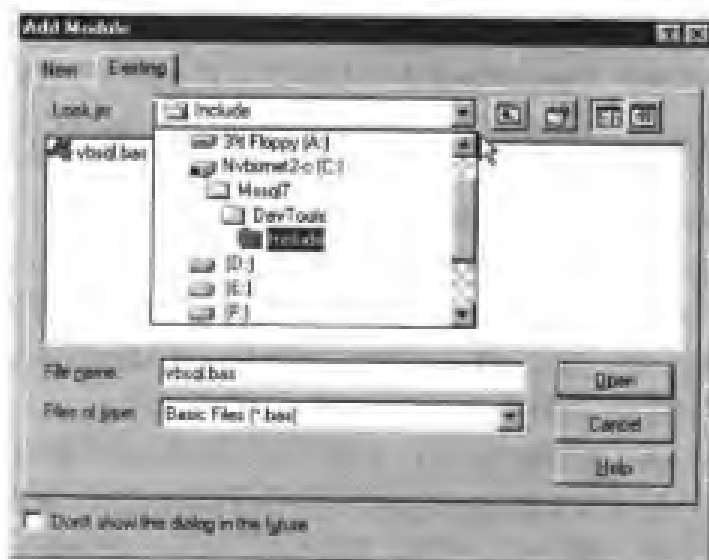


图 15.12 Visual Basic (VB) 的 Add Module 对话框

2. 单击 Existing 选项卡并在 File_name（文件名）域中键入 VBSQL.BAS 模块的全路径。如果（例如）将 MS-SQL Server 开发工具安装在 C:\MSSQL7 文件夹中，则可在 File_name 域内键入 C:\MSSQL7\DEVTOOLS\VBSQL.BAS。

3. 单击 Open 按钮。VB 将把 VBSQL.BAS 模块添加到当前工程中并返回 VB 应用程序窗口。

既然在工程中有了 Vbsql 控件和 VBSQL.BAS 模块,则可使用 DBLIB 函数来处理 MS-SQL Server 数据。

技巧 299 使用 SqlInit()函数初始化 DB 函数库以及使用 SqlWinExit 例程释放由 SqlInit()分配的内存

DB 函数库 (DBLIB) 包括允许 VB 应用程序处理 MS-SQL Server 数据库上的数据的函数和子程序。但是,在能够调用 DBLIB 中的任何其他例程之前,必须先调用 SqlInit。当运行在 Windows 环境时,DBLIB 保存着每个引用它的应用程序的有关信息。调用 SqlInit 函数分配内存并初始化 DBLIB 使用的变量,以便跟踪特定应用程序对 DBLIB 的使用。

下面的代码调用 SqlInit 初始化 DBLIB,然后调用 SqlWinExit 释放由 DBLIB 为应用程序分配的内存资源:

```
Sub main()  
  
    DIM sDBLIBVersion AS String  
  
    sDBLIBVersion = SqlInit()  
    If sDBLIBVersion = vbNullString Then  
        MsgBox "Error! Failed to initialize the DB-Library!"  
        Exit Sub  
    Else  
        'Make subroutine calls and start the main program  
        'loop here  
        MsgBox sDBLIBVersion  
        'Exit the DB-Library and free its memory resources  
        SqlWinExit  
    End If  
End Sub
```

查看上面的 VB 的示例代码之后,读者或许会认为此程序除了使人了解是否能够成功地初始化 DBLIB 而没有做任何事情。”——没错!正如在技巧 300~技巧 324 中的工程所表明的,以后可向 MAIN()过程的 If-Then-Else 结构的 Else 部分添加代码。

现在,要理解的重要事情是,VB 应用程序在调用任何其他例程之前,必须调用 SqlInit 函数。SqlInit 函数没有参数,如果不能为 VB 应用程序初始化 DBLIB,则返回 NULL 字符串。如果成功,SqlInit 返回一个字符串,其中包括 DBLIB 的版本标识。如果 SqlInit 返回空字符串,一定不要调用其他 DBLIB 例程,因为这样做将引起不可预测的结果。

正如编写得好的程序的情况一样,示例中的应用程序在退出之前释放分配给它的内存资源。在这种情况下,SqlWinExit 函数调用告诉 DBLIB 释放由 SqlInit 分配的内存。SqlWinExit 没有参数也不返回值。

在调用了 SqlWinExit 例程之后,应用程序必须再次调用 SqlInit,然后才能接着调用 DBLIB 内的其他例程或函数。

注意:在编译和执行示例程序之前,必须在 VB 程序的一个窗体(如 Form1)上添加 Vbsql 控件。否则,VB 编译器将以以下错误信息中止执行:“File not found: VBSQL.OCX.”(没有找到

到 VBSQLOCX 文件。) 为了向窗体上添加 Vbsql, 请执行以下步骤:

1. 双击工程窗口中的 Form1。VB 将在其设计窗口中显示 Form1。
2. 单击 VB 工具箱上的 Vbsql 控件按钮。

3. 在 Form1 上拖动鼠标画出 Vbsql 控件区域。控件的尺寸和位置并不重要, 因为 VB 应用程序并不在屏幕上显示 Form1。

技巧 300 使用 SqlOpenConnection()函数登录 MS-SQL Server

在应用程序成功调用 SqlInit 函数初始化 DBLIB 之后, 在处理数据之前, 程序必须登录到 MS-SQL Server 上。登录过程相当直接, 因为只涉及一个函数的调用, 即 SqlOpenConnection, 其句法如下:

```
hConnHandle = SqlOpenConnection(sServerName, sLoginID,  
                                sPassword, sWsName, sAppName)
```

其中:

- Long hConnHandle 是由 SqlOpenConnection 函数返回的连接句柄。应用程序将在下面的 DBLIB 函数调用中使用 hConnHandle 的值。接下来的函数调用通过由 SqlOpenConnection 函数打开的连接向 MS-SQL Server 发送命令。
- String sServerName 是程序想要连接其上的 MS-SQL Server 的名称。
- String sLoginID 是程序的登录 ID, 即其在 MS-SQL Server 上的用户名。sLoginID 可以多达 30 个字符长, 而且应用程序必须向打开的与 MSSQL Server 的连接提供合法的 sLoginID/sPassword (用户名/密码) 对。
- String sPassword 是与 sLoginID 中的用户名 (登录 ID) 关联的密码。sPassword 可以多达 30 个字符长, 而且应用程序必须向打开的与 MSSQL Server 的连接提供合法的 sLoginID/sPassword (用户名/密码) 对。
- String sWsName 是应用程序运行其上的工作站名称。虽然 sWsName 可以多达 30 个字符长, 但 MS-SQL Server 只保存前 10 个字符 (在执行存储过程 SP_WHO 时, MS-SQL Server 将 sWsName 显示为 HOSTNAME)。
- String sAppName 是 MS-SQL Server 在其 SYSPROCESSES 表中放置的名称, 以便帮助识别连接。虽然 sAppName 可以多达 30 个字符长, 但 MS-SQL Server 只保存前 16 个字符 (sAppName 是可选的, 可使其为空白)。

如果 SqlOpenConnection 在与名为 sServerName 的 DBMS 连接效果上是成功的, 连接句柄 (hConnHandle) 的值将是非零的。如果不成功, SqlOpenConnection 函数返回的连接句柄值为 0, 因而在使用此句柄在接下来的 DBLIB 函数调用中访问 DBMS 数据之前一定要测试 hConnHandle 的值。

例如, 为了打开与名为 NVBizNet2 的 MS-SQL Server 的连接, 以用户名 konrad、密码 king 登录, 程序要做如下的 SqlOpenConnection 函数调用:

```
hConnHandle = SqlOpenConnection("NVBizNet2", _  
                                "konrad", "king", "myworkstation", _  
                                App.EXENAME)
```

技巧 301 使用 SqlClose()例程关闭单个 MS-SQL Server 连接或者调用 SqlExit 关闭所有打开的连接

正如读者在技巧 300 “使用 SqlOpenConnection()函数登录 MS-SQL Server”中所学习的, 可以调用 SqlOpenConnection 登录 MS-SQL Server。SqlClose 具有相反的效果——它结束(关闭)单个的与 MS-SQL Server 的连接。由于程序可打开多个与一个或多个 MSSQL Servers 的连接, SqlClose 例程的句法如下:

```
SqlClose (hConnHandle)
```

其中包括一个参数, 即想要关闭的连接的连接句柄 (hConnHandle)。

例如, 如果应用程序执行了以下函数调用:

```
hConnHandle1 = SqlOpenConnection("NVBizNet2", _  
    "konrad", "king", "my_ws-1", App.EXENAME)  
hConnHandle2 = SqlOpenConnection("NVBizNet2", _  
    "konrad", "king", "my_ws-2", App.EXENAME)  
hConnHandle3 = SqlOpenConnection("NVBizNet2", _  
    "konrad", "king", "my_ws-3", App.EXENAME)
```

应做如下子程序调用来关闭第二个连接, 而使第一个和第 3 个连接仍然打开供接下来的 DBLIB 函数调用使用:

```
SqlClose(hConnHandle2)
```

作为一种关闭每个连接的替代方法, 应用程序可调用 SqlExit 例程关闭所有打开的连接:

```
SqlExit
```

请注意, SqlExit 没有参数。因而, 在本例中, 如果在调用 SqlClose(hConnHandle2)之后执行, 则调用 SqlExit 一次将关闭其余打开的连接 (hConnHandle1 和 hConnHandle3)。当然, 如果程序还没有调用 SqlClose, 那么调用 SqlExit 将关闭所有 3 个打开的连接。

技巧 302 使用 SqlSendCmd 函数向 MS-SQL Server 发送用于执行的 SQL 语句

DBLIB 提供两种向 MS-SQL Server 发送要执行的 SQL 语句的方法。SqlSendCmd 允许发送单条命令。另一方面, 调用 SqlCommand 允许将多条 SQL 语句放入批处理中, 然后通过调用 SqlExec 函数告诉 DBMS 执行(读者将在技巧 304 “使用 SqlColName()函数提取由查询生成的结果集中的列名”中学习如何与 SqlCommand 结合使用 SqlExec)。

为了使用 SqlSendCmd 函数把 SQL 语句发送到 MS-SQL Server, 可使用以下句法:

```
nRetCode = SqlSendCmd(hConnHandle, sSQLStatement)
```

其中:

- Long nRetCode 是由 SqlSendCmd 函数调用返回的整数结果。在函数调用之后, nRetCode 的值既可是 SUCCEED (1) 也可是 FAIL (0)。
- Long hConnHandle 是由 SqlOpenConnection 函数返回的连接句柄。
- sSQLStatement 是字符串(变量)其中包括 MS-SQL Server 要执行的 SQL 语句。

例如, 为了执行一条 USE 语句将默认数据库切换为 NORTHWIND, 然后执行 UPDATE 语句增加 PRODUCTS 表的 UNITPRICE 列值, 可执行以下两条 SqlSendCmd 函数调用:

```
nRetCode = SqlSendCmd(hConnHandle, "USE Northwind")  
nRetCode = SqlSendCmd(hConnHandle, _
```

```
"UPDATE PRODUCTS SET unitprice = unitprice * 1.20")
```

在调用 `SqlSendCmd` 函数之前，当然应用程序必须调用 `SqlOpenConnection` 打开与目标 MS-SQL Server 的连接（正如读者在技巧 300 “使用 `SqlOpenConnection()` 函数登录 MS-SQL Server” 中所做的）。

技巧 303 使用 `SqlNumCols()` 函数确定由查询生成的结果集中的列数

正如读者在技巧 302 “使用 `SqlSendCmd` 函数向 MS-SQL Server 发送用于执行的 SQL 语句” 中所学习的，可以使用 `SqlSendCmd()` 函数向 DBMS 发送用于执行的 SQL 语句（以字符串的形式）。任何程序用于打开与服务器连接的合法 SQL 语句，只要用户名/登录 ID 具有足够的执行权限，执行起来都非常简单的。因此除了修改数据库中的数据值或对象之外，应用程序可发送返回 0 行、一行或多行数据用于显示或修改的查询。

在应用程序向 MS-SQL Server 发送 SELECT 语句时，DBMS 自动地创建游标并用查询的结果表加以填充。然后应用程序从游标中将数据值提取到程序变量或窗体的域中用于显示或修改。通过将 `SqlNumCols()` 函数与 `SqlColName()` 函数结合使用（参见技巧 304 “使用 `SqlColName()` 函数提取由查询生成的结果集中的列名”），可以建立通用目的的子程序，用查询提取的列名填充 `MSFlexGrid` 控件（读者已经在技巧 278 “向 Visual Basic (VB) 窗体中添加 `MSFlexGrid` 控件以显示 SQL 表数据” 中学习了有关内容）。接着（正如读者将在技巧 305 “使用 `SqlData()` 函数从游标中将查询结果提取到应用程序中” 中学习的），随着调用 `SqlNextRow()` 函数在游标中从一行移动到下一行，可使用 `SqlData()` 函数将游标数据提取到 `MSFlexGrid` 列中。

`SqlNumCols()` 函数调用的句法如下：

```
SqlNumCols (nConnHandle)
```

其中 `Long nConnHandle` 是连接句柄（由 `SqlOpenConnection()` 或 `SqlOpen()` 函数所返回的），此句柄在通过 `SqlSendCmd()`（或 `SqlSend()`）向 MS-SQL Server 发送用于执行的 SQL 语句时已经使用过。

函数返回 `nConnHandle` 游标的当前结果集中的列数。

例如，以下 Visual Basic 程序靠近底部的 `SqlNumCols()` 函数将显示由 DBMS 在执行查询后创建的游标中的列数：

```
Sub main()  
  
Dim sDBLIBVersion As String  
Dim nConnHandle As Long  
Dim nRetCode As Long  
sDBLIBVersion = SqlInit()  
nConnHandle = SqlOpenConnection("NVBizNet2", "konrad", _  
                                "king", "my_ws-1", App.EXENAME)  
nRetCode = SqlSendCmd(nConnHandle, "USE Northwind")  
nRetCode = SqlSendCmd(nConnHandle, "SELECT * FROM products")  
  
MsgBox "Cursor column count = " & SqlNumCols(nConnHandle)  
  
SqlExit
```

```
SqlWinExit
```

```
End Sub
```

其中执行的查询是：

```
SELECT * FROM products
```

当读者在技巧 304 中学习了如何编写通用目的的子程序在 MSFlexGrid 中显示游标列名之后，能够提取游标中列数的用处变得很明显。

注意：为了清楚起见，在本技巧中的示例 VB 程序中没有用于检查诸如确保 SqlInit() 函数返回非 NULL 字符串（指明成功地初始化 DBLIB）以及确保 nConnHandle 不等于 0（指明成功地与 MS-SQL Server 连接）的错误处理程序。

技巧 304 使用 SqlColName() 函数提取由查询生成的结果集中的列名

在技巧 303 “使用 SqlNumCols() 决定由查询生成的结果集中的列数”中，读者学习了如何使用 SqlNumCols() 函数来确定通过调用 SqlSendCmd() 函数向 MS-SQL Serve 发送的查询，DBMS 放入查询的结果集中的列数。能够确定游标中的列数，而程序仍在运行，使得编写单个子程序来提取并显示由不同查询返回的列名成为可能，即使每个查询的 SELECT 子句包括不同数目的列也可以。

SqlColName() 函数调用的句法如下：

```
SqlColName(nConnHandle, iColNo)
```

其中：

- Long nConnHandle 是连接句柄（由 SqlOpenConnection() 或 SqlOpen() 返回的），此句柄在通过 SqlSendCmd()（或 SqlSend()）向 MS-SQL Server 发送用于执行的 SQL 语句（或一组语句）时已经使用过。
- Integer iColNo 是函数提取的列名。游标中的第一列的号码为 1（而不是 0）。

函数返回包括由 iColNo 值引用的结果表列的名称（如果列没有名称或是如果 iColNo 的值大于游标中的列数，函数将返回空字符串）。

因而，以下子程序将填充 MSFlexGrid 对象的第一行（此内容已在技巧 278“向 Visual Basic (VB) 窗体中添加 MSFlexGrid 控件以显示 SQL 表数据”中学习过）：

```
Private Sub GetColumnNames _
    (nConnHandle As Long, FlexGrid As MSFlexGrid)
    Dim i As Integer

    FlexGrid.Cols = SqlNumCols(nConnHandle)
    FlexGrid.Rows = 1
    FlexGrid.Row = 0
    FlexGrid.Col = 0

    For i = 1 To FlexGrid.Cols
        FlexGrid.Text = SqlColName(nConnHandle, i)
        FlexGrid.ColWidth(FlexGrid.Col) = _
            Form1.TextWidth(FlexGrid.Text) + 120
        If FlexGrid.Col < FlexGrid.Cols - 1 Then
            FlexGrid.Col = FlexGrid.Col + 1
        End If
    Next i
End Sub
```

```

End If
Next i

FlexGrid.Redraw = True
End Sub

```

填充使用的数据是作为 nConnHandle 参数传递到子程序中的连接句柄的游标中的查询的结果集中的列名。

技巧 305 使用 SqlData()函数从游标中将查询结果提取到应用程序中

MSFlexGrid 是个“灵活的”数据网格，Visual Basic 程序可用来显示 DBMS 数据，而不必在运行之前了解数据值的列数和行数。正如读者在技巧 304 “使用 SqlColName()函数提取由查询生成的结果集中的列名”中所学习的，一个程序可快速地改变 MSFlexGrid 中的列数。例如，一个应用程序可能在运行的某一时刻将 MSFlexGrid 用于显示 5 列和 300 行的查询结果集，而稍后可将同一网格改变尺寸用于显示每行 15 列共有 250 行的数据。

在向 VB 窗体添加了 MSFlexGrid 之后（参见技巧 278 “向 Visual Basic (VB) 窗体中添加 MSFlexGrid 控件以显示 SQL 表数据”）并填入了列标题（参见技巧 304），可使用 SqlData()函数从连接句柄的游标中将查询结果提取到组成 MSFlexGrid 的单元格中。SqlData()函数调用的句法如下：

```
SqlData(nConnHandle, iColNo)
```

其中：

- Long nConnHandle 是连接句柄（由 SqlOpenConnection()或 SqlOpen()所返回的），此句柄在通过 SqlSendCmd()（或 SqlSend()）向 MS-SQL Server 发送用于执行的 SQL 语句（或一组语句）时已经使用过。
- Integer iColNo 是函数提取数据的列数。游标中的第一列号码是 1（而不是 0）。

函数返回一个包括结果列中的由 iColNo 值给定的数据值的字符串。如果游标列是二进制的数据类型如 varbinary 或 image，SqlData()将返回二进制数据的字符串，其中每字节的游标列中的数据一个字符。对于所有其他数据类型（不管是字符串还是数值），SqlData()将返回可阅读的字符串（如果提取的列数[iColNo]大于最大的游标列数，或者如果游标列是空白的或 NULL，则 SqlData()将返回空的字符串）。

由于每个 SqlData()调用返回只代表游标中一行的单列中数据值的字符串，程序必须调用函数多次才能从有多列的查询结果的行中提取数据值。例如，为了将 nConnHandle 游标中的第 6 列中的数据值提取到 sUnitPrice 字符串中，可使用以下 VB 表达式：

```
sUnitPrice = SqlData(nConnHandle,6)
```

为了从 nConnHandle 游标中将所有列值提取到名为 FlexGrid 的 MSFlexGrid 控件的单元格中，可使用 VB 的 For-Next 环境，如下所示：

```

FlexGrid.col = 0
For i = 1 to SqlNumCols(nConnHandle)
    FlexGrid.text = SqlData(nConnHandle,i)

    If FlexGrid.col < FlexGrid.cols - 1 Then
        FlexGrid.col = FlexGrid.col + 1
    End If
Next i

```

```
End If
Next i
```

技巧 306 使用 SqlNextRow()函数在游标行中向前移动

正如读者在技巧 305 中所学习的, SqlData()函数从游标的当前行的一列中提取数据值。SqlNextRow()函数通过把游标指针一次向前移动一行,从而改变游标中的当前行。由于 DBMS 在用查询结果填充游标之后, DBLIB 将游标指针定位于第一行之前,因而在第一次 SqlData()函数调用从查询结果的第一行中的列中提取数据之前,一定要调用 SqlNextRow()函数。通过不断地调用 SqlNextRow()函数,直到返回 NOMOREROWS 值为止,应用程序可一次一行地通过 DBMS 放入游标的查询结果的所有行。

SqlNextRow()函数调用的句法如下:

```
SqlNextRow (nConnHandle)
```

其中 Long nConnHandle 是连接句柄(由 SqlOpenConnection()或 SqlOpen()所返回的),此句柄在通过 SqlSendCmd() (或 SqlSend()) 向 MS-SQL Server 发送用于执行的 SQL 语句(或一组语句)时已经使用过。

此函数返回以下整数值之一:

- REGROW (-1)。如果游标的当前行包括 SELECT 语句结果的话。
- <COMPUTE clause ID number>。如果游标的当前行包括来自 COMPUTE 语句的结果的话。
- FAIL (0)。如果 SqlNextRow()函数不成功的话。
- NOMOREROWS (-2)。如果游标中不再有行的话——不管是 SELECT 未返回行还是 DBMS 由于死锁不能执行查询。
- BUFFULL (-3)。如果启用了行缓冲而且缓冲区已满。

例如, 以下 Visual Basic 子程序:

```
Private Sub GetResultsSet _
    (nConnHandle As Long, FlexGrid As MSFlexGrid)
    Dim i As Integer

    FlexGrid.Redraw = False
    FlexGrid.AllowUserResizing = FlexResizeColumns
    FlexGrid.FixedCols = 0
    FlexGrid.Row = 0
    Do Until NOMOREROWS = SqlNextRow(nConnHandle)
        FlexGrid.Col = 0
        FlexGrid.Rows = FlexGrid.Rows + 1
        FlexGrid.Row = FlexGrid.Row - 1

        For i = 1 To FlexGrid.Cols
            FlexGrid.Text = SqlData(nConnHandle,i)
            If FlexGrid.Col < (FlexGrid.Cols - 1) Then
                FlexGrid.Col = FlexGrid.Col + 1
            End If
        Next i
    Loop
```

```
Next i
Loop

FlexGrid.Redraw = True
End Sub
```

将来自 nConnHandle 游标的查询结果的行中的每列填充 MSFlexGrid 对象的单元格行（读者已在技巧 278 “向 Visual Basic (VB) 窗体中添加 MSFlexGrid 控件以显示 SQL 表数据”中学习过有关内容）。

注意：本例中的子程序假定程序是在调用了 GetColumnNames 子程序（参见技巧 304 “使用 SqlColName() 函数提取由查询生成的结果集中的列名”）之后调用的。此子程序设置了 MSFlexGrid 的列数属性（FlexGrid.Cols）并将行数属性（FlexGrid.Rows）设置为从 1 开始（以考虑网格中的列标题行）。

技巧 307 使用 SqlCommand() 函数建立 SQL 语句批处理

虽然 SqlSendCmd() 函数（读者已在技巧 302 “使用 SqlSendCmd 函数向 MS-SQL Server 发送用于执行的 SQL 语句”中学习了有关内容）允许人们向 MS-SQL Server 发送用于执行的单条 SQL 语句，但 SqlCommand() 函数允许建立语句批处理。在完成向批处理中添加语句之后，SqlExec() 和 SqlSend() 函数允许把整个一组语句发送到 MS-SQL Server，像单条语句一样去执行。

SqlCommand() 函数调用的句法如下：

```
nRetCode = SqlCommand(nConnHandle, sSQLStatement)
```

其中：

- Long nRetCode 是由 SqlCommand() 函数调用返回的整数结果。在函数调用之后，nRetCode 的值既可为 SUCCEED (1) 也可 FAIL (0)。
- Long nConnHandle 是由 SqlOpenConnection() 或 SqlOpen() 函数返回的连接句柄。
- String sSQLStatement 是函数追加到已有的 nConnHandle 命令缓冲区内容上的字符串。SqlCommand() 函数调用中的 sSQLStatement 参数可包括完整的 SQL 语句，也可包括一个或多个语句的一部分。

例如，下面的 Visual Basic 代码：

```
nRetCode = SqlCommand(nConnHandle, "USE Northwind")
nRetCode = SqlCommand(nConnHandle, " SELECT * FROM products")
nRetCode = SqlCommand(nConnHandle, " USE Pubs")
nRetCode = SqlCommand(nConnHandle, " SELECT * FROM authors")
```

建立包括两条 USE 语句和两条查询的 SQL 语句批处理。每个 SqlCommand() 函数调用都向 sSQLStatement 参数已有的 nConnHandle 语句缓冲区的内容中追加 sSQLStatement 参数中的内容（不删除和改写任何已有的）（连接句柄的语句缓冲区并不清除，直到程序调用了 SqlExec() 或 SqlSend() 函数把缓冲区内容发送到 DBMS 用于执行为止）。

注意：每个 SqlCommand() 函数调用都将语句缓冲区的内容与 sSQLStatement 参数中的内容串接起来。因而，在向 SQL 语句批处理中添加语句时，一定要在每个语句之后或是在第二条语句的第一个字符之前添加至少一个空格。例如，请注意一下本技巧的示例中的每条语句的开始都有空格。

技巧 308 使用 SqlExec()函数将 SQL 语句批处理提交给 MS-SQL Server 执行

正如读者在技巧 307 “使用 SqlCmd()函数建立 SQL 语句批处理”中所学习的, SqlCmd()函数调用允许创建 SQL 语句批处理, 将字符串追加到连接句柄命令缓冲区中已有的文本上, 调用 SqlExec()函数将命令缓冲区的内容 (SQL 语句批处理) 作为单条字符串 (或许有点长) 发送到 DBMS 执行。

SqlExec()函数调用的句法如下:

```
nRetCode = SqlExec(nConnHandle)
```

其中:

- Long nRetCode 是由 SqlExec()函数调用返回的整数结果。其值既可为 SUCCEED (1) 也可为 FAIL (0)。
- Long nConnHandle 是由 SqlOpenConnection()或 SqlOpen()函数返回的连接句柄。

例如, 如果在技巧 307 中, 以下 Visual Basic 表达式跟在 SqlCmd()函数调用的后面:

```
nRetCode = SqlExec(nConnHandle)
```

则 SqlExec()函数调用将向 MS-SQL Server 发送以下字符串并清除命令缓冲区并等待 DBMS 完成批处理中的所有语句的执行:

```
USE Northwind SELECT * FROM products USE Pubs SELECT * FROM authors
```

如果命令缓冲区是空的, 或者如果任何语句包括句法错误或试图执行超过其连接权限的行动, SqlExec()函数将返回值为 FAIL (0)。作为对照, 如果 DBMS 成功地执行了所有批处理中的语句, SqlExec()返回值为 SUCCEED (1), 而每条语句的结果都放在连接句柄的游标中, 通过调用 SqlResults()函数可供提取 (读者将在技巧 309 “使用 SqlResults()函数提取 SqlExec()发送的查询结果集”中学习)。

注意: 由 SqlExec()函数调用发送到 MS-SQL Server 的语句批处理中的一组 SQL 语句可看作是单条事务处理。因而, 如果 DBMS 不能成功执行批处理中的任何一条语句, 系统将取消批处理中前面语句所完成的工作, 从而不返回查询的任何行。

技巧 309 使用 SqlResults()函数提取 SqlExec()发送的查询结果集

在调用 SqlExec()函数将 SQL 语句批处理发送到 MS-SQL Server 执行之后, 应用程序必须对批处理中的每条语句调用 SqlResults()函数一次, 从而告诉 DBLIB 准备每条语句的结果集, 以备应用程序中的函数调用提取数据。另外, 如果忘记调用 SqlResults()函数, 直到再没有未准备的结果集, 则接下来的 SqlExec()调用都会失败, 因为前面的结果还没有处理完成。虽然不必调用 SqlData()提取由传送语句批处理的 SqlExec() (或 SqlSend()) 内的 SELECT 语句 (如果有的话) 返回的所有数据值, 但确实必须通过调用 SqlResults()函数将每个结果集标记为准备好的和处理过的, 直到再没有用于处理的结果集为止。

注意: 应用程序必须对由 SqlExec()或 SqlSend()函数发送的 SQL 语句批处理中的每条语句调用 SqlResults()函数一次——而不管语句是否返回数据行。例如, 虽然在技巧 307 “使用 SqlCmd()函数建立 SQL 语句批处理”中创建的语句批处理并在技巧 308 “使用 SqlExec()函数将 SQL 语句批处理提交给 MS-SQL Server 执行”发送到 MS-SQL Serve, 批处理中包括两条 USE 语句 (执行时并不返回数据行) 和两条 SELECT 语句 (可能返回数据行), 程序必须在向 MS-SQL Server 发送了示例中的 SQL 语句批处理用于执行之后, 调用 SqlResults()4 次。

SqlResults()函数调用的句法如下:

```
nRetCode = SqlResults(nConnHandle)
```

其中:

- Long nRetCode 由 SqlResults()函数调用返回的整数结果。其值可为 SUCCEED (1)、FAIL (0)、NOMORERESULTS(2)或 NOMORERPCRESULTS (3)。
- Long nConnHandle 是由 SqlOpenConnection()或 SqlOpen()函数返回的连接句柄。此句柄由 SqlSend()用来向 MS-SQL Server 发送用于执行的语句批处理。

为了避免不得不跟踪命令批处理中的语句命令数,从而了解要调用多少次 SqlResults()函数,只要使用如下的 Do-Until 循环即可:

```
Do Until NOMORERESULTS = SqlResults(nConnHandle)
  If SqlNumCols(nConnHandle) > 0 Then
    'Routines that retrieve and work with query results
    GetColumnNames nConnHandle, Form1.QueryResults
    GetResultsSet nConnHandle, Form1.QueryResults
    MsgBox "Click your mouse pointer on the OK button " _
      & "to process the next set of results."
  End If
Loop
```

这将不断地调用 SqlResults()函数,直到再没有要处理的结果集为止(此时 SqlResults()将返回 NOMORERESULTS 值)。

由于 SqlResults()函数只准备结果集用于进一步处理,应用程序必须调用 SqlNextRow()函数移过查询结果集游标中的所有行,而用 SqlData()函数提取游标当前行中的列值。例如,在前例中的 Do-Until 循环中检查了执行语句的结果集,看是否返回了数据(在 SqlNumCols(nConnHandle) > 0 的情况下)。如果返回了,那么程序调用该子程序(在技巧 304 “使用 SqlColName()函数提取由查询生成的结果集中的列名”和技巧 306 “使用 SqlNextRow()函数在游标行中向前移动”中开发出的子程序)提取游标列名和数据值。

技巧 310 使用 SqlSend()提交语句批处理而不必等待 DBMS 完成所有语句的执行

正如在技巧 308 “使用 SqlExec()函数将 SQL 语句批处理提交给 MS-SQL Server 执行”中所提到的,一条 SqlExec()函数调用在 SqlExec()函数等待 DBMS 完成其对 SQL 语句批处理的处理时,其他程序的执行都得停止。因此,如果批处理中的一条(或多条)语句提取大量数据或是对大量数据排序,或者等待另一位 DBMS 用户释放所需的资源,则应用程序看起来就好像是挂起一段时间一样。如果在 DBMS 执行批处理中的 SQL 语句时想让程序执行其他任务,可调用 SqlSend()函数来提交批处理,而不要调用 SqlExec()函数。

SqlSend()函数调用的句法如下:

```
nRetCode = SqlSend(nConnHandle)
```

其中:

- Long nRetCode 是由 SqlSend()函数调用返回的整数结果。在函数调用之后, nRetCode 的值可为 SUCCEED (1)或 FAIL (0)。
- Long nConnHandle 是由 SqlOpenConnection()或 SqlOpen()返回的连接句柄。

在执行如下的表达式时:

```
nRetCode = SqlSend(nConnHandle)
```

(此表达式告诉 DBLIB 向 MS-SQL Server 发送 nConnHandle 命令缓冲区中的 SQL 语句), 系统将执行程序中的下一条语句——而不必等待 DBMS 完成批处理的语句处理。

但是, 在进行另一 SqlSend()或 SqlExec()函数调用之前, 必须调用 SqlOk()函数, 看 DBMS 是否完成了批处理中的语句的处理。事实上, 以下的函数调用:

```
nRetCode = SqlExec(nConnHandle)
```

与下面的函数调用是等价的:

```
nRetCode = SqlSend(nConnHandle)
```

```
nRetCode = SqlOk(nConnHandle)
```

由于 SqlOk() (与 SqlExec()类似) 将等待 DBMS 完成 nConnHandle 命令缓冲区的语句处理, 以后才能将控制权返回给应用程序。

幸运的是, DBLIB 允许使用 SqlDataReady()函数 (读者将在技巧 311 “使用 SqlDataReady()函数确定 MS-SQL Server 是否完成了 SQL 语句批处理” 中学习有关内容) 检查 DBMS 是否已经完成了语句批处理的执行任务 (由 SqlSend()提交的), 而不必等待系统实际上完成批处理中的所有语句的执行。

因此, 一个允许应用程序在等待 DBMS 处理批处理中的 SQL 语句时执行其他任务的 Visual Basic 命令序列如下所示:

```
'SqlCmd() function calls to build the SQL batch
SqlSend(nConnHandle)

Do Until SUCCEED = SqlDataReady(nConnHandle)
  'VB statements that perform work you want the application
  'to do while waiting for the DBMS to finish processing
  'the SQL statements in the 'nConnHandle' command buffer
Loop
If SqlOk(nConnHandle) = SUCCESS Then
  'VB statements that process the results sets (one for
  'each statement in the batch) for the SQL statement batch
  'submitted to the MS-SQL Server by the SqlSend() function
  'call
End If
```

技巧 311 使用 SqlDataReady()函数确定 MS-SQL Server 是否完成了 SQL 语句批处理

在技巧 310 “使用 SqlSend()提交语句批处理而不必等待 DBMS 完成所有语句的执行” 中, 读者学习了如何使用 SqlSend()函数向 MS-SQL Server 提交用于处理的连接句柄的命令缓冲区的内容。由调用 SqlSend()而不是 SqlExec()所获得的主要好处是, SqlSend()函数向 MS-SQL Server 提交用于处理的连接句柄的命令缓冲区的内容, 并允许在 DBMS 执行批处理中的语句时应用程序继续其他工作 (正如读者在技巧 308 “使用 SqlExec()函数将 SQL 语句批处理提交给 MS-SQL Server 执行” 中所学习的, 在应用程序调用 SqlExec()之后, SqlExec()函数将连接句柄的命令缓冲区的内容提交给 MS-SQL Server 用于处理, 同时在等待 DBMS 执行语句批处理中的所有语句时停止程序的进一步执行)。

SqlDataReady()函数允许检查 DBMS 是否完成了由调用 SqlSend()函数提交给 MS-SQL Server 的语句批处理。在应用程序调用 SqlSend()之后（而且 DBLIB 将命令缓冲区的内容发送给 DBMS），程序必须调用 SqlOk()查看 DBMS 是否成功地执行了批处理中的所有语句。遗憾的是，SqlOk()与 SqlExec()一样，直到 DBMS 完成处理全部批处理之后才将控制权返回应用程序。

另一方面，SqlDataReady()并不等待 DBMS 完成对命令缓冲区内容的处理，而只是检查 DBMS 是否完成了批处理。因而，如果 SqlDataReady()返回 SUCCEED (1)，调用 SqlOk()不会引起程序“挂起”，因为 DBMS 已经完成了批处理。结果，SqlOk()可提取批处理结果代码，而不必等待。另一方面，如果 SqlDataReady()返回 FAIL (0)，DBMS 还未完成批处理，而应用程序应该在稍后调用 SqlOk()，以便给 DBMS 以完成执行批处理中其余语句的机会。

SqlDataReady()函数调用的句法如下：

```
nRetCode = SqlSend(nConnHandle)
```

其中：

- Long nRetCode 是由 SqlDataReady()函数调用返回的整数结果。在函数调用之后，nRetCode 的值既可是 SUCCEED (1)（意味着 DBMS 已经完成执行批处理中的语句），也可为 FAIL (0)（意味着 DBMS 仍在执行批处理中的语句）。
- Long nConnHandle 是由 SqlOpenConnection()或 SqlOpen()函数返回的连接句柄。此句柄由 SqlSend()用来向 MS-SQL Server 发送用于执行的语句批处理。

例如，Visual Basic 工程 PROJECT405.VBP 中的 Form1 窗体上的 Check if Done 按钮使用的如下代码：

```
Private Sub DataReady_Click()  
If SqlDataReady(nConnHandle) = SUCCEED Then  
If SqlOk(nConnHandle) = SUCCEED Then  
GetBatchResults (nConnHandle) .  
Else  
MsgBox "The statement batch failed to execute."  
End If  
Else  
MsgBox "The DBMS is still processing the batch."  
End If  
End Sub
```

此段代码让用户检查语句批处理的进度并仅当 DBMS 完成批处理之后才调用 SqlOk()一次，这就允许在 DBMS 执行 SQL 语句批处理时，用户还可做其他工作。

技巧 312 使用 SqlCancel()终止发送到 MS-SQL Server 的语句批处理并清除批结果缓冲区

SqlCancel()函数允许应用程序告诉 MS-SQL Server 终止处理语句批处理。当 DBMS 收到 SqlCancel()函数的取消请求时，它终止批中当前语句的执行，撤消由已经部分执行的语句批处理所完成的所有工作，并清除批结果缓冲区。

SqlCancel()函数调用的句法如下：

```
nRetCode = SqlCancel(nConnHandle)
```

其中：

- Long nRetCode 是由 SqlCancel()函数调用返回的整数结果。在函数调用以后, nRetCode 的值将变为 SUCCEED (1) (意思是 DBMS 能响应取消请求)或 FAIL (0)。
- Long nConnHandle 是 SqlSend()、SqlExec()或 SqlSendCmd()用来向 MS-SQL Server 发送一个或多个供执行的 SQL 语句的连接句柄(由 SqlOpenConnection()或 SqlOpen()函数返回)。

因此, Visual Basic (VB) 程序中的表达式:

```
nRetCode = SqlCancel(nConnNVBizNet2)
```

将告诉由 nConnNVBizNet2 连接句柄连接的 MS-SQL Server 应用程序终止连接的语句批处理并从连接结果缓冲区清除未决的结果。

除了终止语句批处理的执行(通过在调用 SqlSend()后调用 SqlCancel())外, 应用程序还可以在调用 SqlOk()、SqlExec()或 SqlSendCmd()后通过调用 SqlCancel()从已执行的语句批处理清除未决结果。正如已经在技巧 309 “使用 SqlResults()函数提取 SqlExec()发送的查询结果集”中学习过的, 程序必须反复调用 SqlResults()函数(语句批处理中的每个语句一次), 直到不再有结果集要处理。然而, 如果不需要批处理中的语句产生的结果或者语句不产生任何查询结果, 可以通过调用 SqlCancel()函数来跳过 SqlResults()函数, 这会清除所有的未决结果。

技巧 313 使用 SqlCanQuery()函数在当前结果集中删除剩余(未被处理的)行

虽然 SqlCancel()函数允许停止语句批处理并放弃所有结果集, 但 SqlCanQuery()允许放弃当前结果集中未被处理的结果的剩余行。例如, 假设 Visual Basic (VB) 程序执行下面的表达式:

```
nRetCode = SqlSendCmd(nConnHandle, "USE Northwind" _  
& " SELECT * FROM products USE Pubs " _  
& " SELECT * FROM authors "
```

该语句向 MS-SQL Server 发送 4 个 SQL 语句供执行。正如已经在技巧 309 “使用 SqlResults()函数提取 SqlExec()发送的查询结果集”中学习过的, 程序必须在再次调用 SqlSendCmd()、SqlSend()或 SqlExec()前调用 SqlResults()函数 4 次(每个发送到 MS-SQL Server 的语句一次)。而且, 正如在技巧 306 “使用 SqlNextRow()函数在游标行中向前移动”中学习过的, 在这个示例中每个由这两个查询返回的行, 程序都必须调用 SqlNextRow()函数一次。如果直到函数返回值为 NOMOREROWS 时, 应用程序才调用 SqlNextRow(), 后面对 SqlResults()函数的调用就会失败, 因为当前结果集(通过调用 SqlResults()提取)中所有的行都没有处理(通过调用 SqlNextRow())。

如果想停止处理 DBMS 执行以下查询时生成的结果集中的行:

```
SELECT * FROM products
```

而不必对查询结果表中 PRODUCTS 的每一行都调用 SqlNextRow()函数, 就按如下方式调用 SqlCanQuery():

```
nRetCode = SqlCanQuery(nConnHandle)
```

其中:

- Long nRetCode 是由 SqlCanQuery()函数调用返回的整数结果。在函数调用以后, nRetCode 值将变为 SUCCEED (1)(意思是 DBLIB 能清除查询结果中剩余的未被处理的行)或 FAIL (0)。
- Long nConnHandle 是 SqlSend()、SqlExec()或 SqlSendCmd()用来向 MS-SQL Server 发

送一个或多个供执行的 SQL 语句的连接句柄（由 `SqlOpenConnection()` 或 `SqlOpen()` 函数返回）。

DBLIB 将随后放弃任何从 `SELECT * FROM products` 查询获得的查询结果的未被处理的行。然而，从第二个查询 `SELECT * FROM authors` 获得的结果行在调用 `SqlResults()` 函数两次以后将仍然可以提取（通过调用 `SqlNextRow()` 函数）——一次是为 `USE Pubs` 语句准备结果集，第二次是为 `SELECT * FROM authors` 语句准备结果集。

注意：如果 VB 应用程序处理第一个查询（`SELECT * FROM products`）获得的结果集时调用 `SqlCancel()` 函数（而不是调用 `SqlCanQuery()` 函数），则 DBLIB 将不仅会放弃查询结果集的剩余行，也将放弃从剩余的两个 SQL 语句（`USE Pubs` 和 `SELECT * FROM authors`）获得的结果集。

技巧 314 使用 `SqlUse()` 函数为 MS-SQL Server 连接设置当前数据库

`SqlUse()` 函数允许为与 MS-SQL Server 的连接选择当前数据库。当执行 SQL 语句时，MS-SQL Server 假定它将在会话的当前数据库中找到在 `FROM` 子句中引用的表。同样地，如果登录到 MS-SQL Server 且起始的默认数据库是 `NORTHWIND`（以此为例），可以执行下面的语句：

```
SELECT * FROM products
```

因为 `PRODUCTS` 是 `NORTHWIND` 数据库中的一个表。另一方面，如果在当前数据库仍然是 `NORTHWIND` 时试图执行这样的语句：

```
SELECT * FROM authors
```

则 DBMS 将终止查询并显示类似于下面的错误消息：

```
Server: Msg 208, Level 16, State 1, Line1  
Invalid object name 'authors'.
```

既可以通过键入 `AUTHORS` 表的全部合格的完整对象名，如：

```
SELECT * FROM pubs.dbo.authors
```

也可以通过选择新的默认/当前数据库，然后用表未经限定的名称再次提交查询：

```
USE pubs
```

```
SELECT * from authors
```

在 `SqlUse()` 函数调用的以下句法中：

```
nRetCode = SqlUse(nConnHandle)
```

下面的说明是真实的：

- Long `nRetCode` 是由 `SqlUse()` 函数调用返回的整数结果。在函数调用以后，`nRetCode` 值将变为 `SUCCEED (1)` 或 `FAIL (0)`。
- Long `nConnHandle` 是连接句柄（由 `SqlOpenConnection()` 或 `SqlOpen()` 函数返回）。

这个调用执行与 Transact-SQL `USE` 语句相同的工作。所以，Transact-SQL `USE` 应用能够选择 `PUBS` 数据库并通过执行下面的语句查询它的表：

```
nRetCode = {nConnHandle, "PUBS"}
```

```
nRetCode = SqlSendCmd(nConnHandle, "SELECT * FROM authors")
```

注意：当调用 `SqlUse()` 函数的时候，请记住 DBLIB 通过用 `USE <database name>` 字符串填充连接句柄的语句缓冲区并调用 `SqlExec()` 和 `SqlResults()` 函数向 DBMS 发送供执行的 `USE` 语句来实现该函数。同样地，如果调用 `the SqlCmd()` 函数把语句放入连接句柄的命令缓冲区中，

一定要在调用 `SqlUse()` 函数前调用 `SqlCmd()` 函数（或 `SqlSend()` 函数）把供执行的语句批处理发送到 MS-SQL Server, `SqlUse()` 函数将用一个 USE 语句覆盖命令缓冲区的当前内容。而且, 如果在当前结果集中有未被处理的结果集或未被处理的行, `SqlUse()` 函数调用将返回结果代码 FAIL (0), 因为它用 `SqlExec()` 函数向 DBMS 发送该 USE 语句。直到不再有还未被处理的结果集且在当前结果集中不再有未被处理的行的时候, 应用程序才能调用 `SqlExec()` 函数。

技巧 315 使用 `Vbsql1_Error()` 例程显示 DBLIB 生成的错误消息

当将 Visual Basic (VB) 的 SQL 控件 (VBSQL.OCX) 包含到 VB 工程中的一个窗体上之后, 就可以定义 DBLIB 在运行时如果遇到错误时将执行的错误处理例程。虽然当 DBLIB 函数不能成功地完成其工作时, 其典型的返回值是 FAIL (0), 但错误代码并没有说明函数为什么执行失败。幸运的是, 不仅可以使使用 DBLIB 错误处理程序 `Vbsql1_Error()` 显示错误号和错误的严重性, 还可以显示所要进行的工作出了故障的文本消息。

要向 VB 工程添加 DBLIB 错误处理程序, 请执行下列步骤:

1. 在要添加 VBSQL.OCX 控件的窗体上单击。例如, 如果在技巧 298 的工程上工作, 可把 VBSQL.OCX 控件添加到 `Form1` 窗体上。所以, 在 VB 应用程序窗口靠近右上角的工程窗口中的 `Form1.frm` 上单击。
2. 从 Standard (标准) 工具栏选择 View (视图) 菜单中的 Code (代码) 选项。VB 将显示为 `Form1` 定义的子例程和函数的代码, 如图 15.13 所示。



图 15.13 VB 工程带有 `Form1` 的 Code 窗格的 Visual Basic (VB) 应用程序窗口

3. 在 Object (对象) 域 (代码窗口的顶部) 右边的下拉列表按钮上单击, 并从该下拉列表框中选择 `Vbsql1`。
4. 在 Procedure (过程) 域 (就在代码窗口顶部的 Object 域的右边) 右边的下拉列表按钮上单击, 并从该下拉列表框选择 `Error`。VB 将把 `Vbsql1_Error()` 例程的声明添加到 `Form1` 的 Code 窗格。

接着, 添加在运行时遇到 DBLIB 错误的情况下应用程序要执行的代码。例如, 要显示错误代码、严重性以及错误描述, 可向 `Vbsql1_Error()` 例程的定义体添加如下的 `MsgBox` 调用:

```
Private Sub Vbsql1_Error(ByVal SqlConn As Long, _
```

```

ByVal Severity As Long, ByVal ErrorNum As Long, _
ByVal ErrorStr As String, ByVal OSErrNum As Long, _
ByVal OSErrStr As String, RetCode As Long)
Select Case ErrorNum
Case 10007: 'Do not display non-error (info) messages
Case Else
MsgBox "Error Code: " & ErrorNum & vbCrLf & _
"Severity: " & Severity & vbCrLf & _
"Message: " & ErrorStr, vbOKOnly, _
"DBLib Error in - " & App.EXENAME
End Select
End Sub

```

在将当前示例的代码添加到 Vbsql1_Error() 例程的定义体后, VB 应用程序将用一个 Windows 消息框显示在运行时遇到的任何 DBLIB 错误的错误代码及其描述。

技巧 316 使用 Vbsql1_Message() 例程显示 MS-SQL Server 生成的错误消息

除了为在运行时由 DBLIB 遇到的错误定义错误处理程序（正如在技巧 315 “使用 Vbsql1_Error() 例程显示 DBLIB 生成的错误消息”中学习过的）外, 还可以为由 MS-SQL Server 报告给 DBLIB 的错误创建错误处理程序。DBLIB 错误包括这样一些, 如试图在没有调用 SqlResults() 函数的情况下再次调用 SqlExec() 函数, 直到返回 NOMORERESULTS; 或者在调用 SqlNextRow() 函数前调用 SqlResults() 函数来提取第二个结果集, 直到第一次返回 NOMOREROWS。另一方面, MS-SQL Server 报告的错误将包括这样一些内容: 如在 SELECT 子句中包括未定义的对象名的 SELECT 语句, 或当没有为连接设置进行这些操作的正确权限时试图修改对象内容的 UPDATE 语句。简而言之, Vbsql1_Error() 例程报告发生在 DBLIB 内的错误, 而 Vbsql1_Message() 则报告发生在 MS-SQL Server 上的错误。

为了向 Visual Basic (VB) 工程添加 MS-SQL Server 错误消息处理程序, 请执行下列步骤:

1. 在要添加 VBSQL.OCX 控件的窗体上单击。例如, 如果在技巧 298 “向 Visual Basic (VB) 中添加 DB 函数库 (DBLIB) 功能”的工程中工作, 可把 VBSQL.OCX 控件添加到 Form1 窗体, 所以就要在 VB 应用程序窗口靠近右上角的工程窗口中的 Form1.frm 上单击。
2. 从 Standard 工具栏选择 View 菜单中的 Code 选项。VB 将显示为 Form1 定义的子例程和函数的代码。
3. 在 Object 域 (代码窗口的顶部) 右边的下拉列表按钮上单击, 并从该下拉列表框选择 Vbsql1。
4. 在 Procedure 域 (就在代码窗口顶部的 Object 域的右边) 右边的下拉列表按钮上单击, 并从该下拉列表框选择 Message。VB 将把 Vbsql1_Message() 例程的声明添加到 Form1 的 Code 窗格。

接着, 添加只要 MS-SQL Server 在运行时报告 DBLIB 错误的情况下应用程序就要执行的代码。例如, 要显示消息代码、错误状态、严重性以及错误描述, 可向 Vbsql1_Message() 例程的定义体添加如下的 MsgBox() 调用:

```

Private Sub Vbsql1_Error(ByVal SqlConn As Long, _
ByVal Message As Long, ByVal State As Long, _
ByVal Severity As Long, ByVal MsgStr As String, _

```



```

ByVal ServerNameStr As String, ProcNameStr As String, _
ByVal Line As Long)
Select Case Message
Case 5701: 'Do not display non-error (info) messages
Case Else
MsgBox "Error reported by MS-SQL Server: " & _
ServerNameStr & "." & vbCrLf & "Msg: " & _
Message & ", Severity: " & Severity & _
", State: " & State & ", Line: " & _
Line & vbCrLf & "Message: " & MsgStr, vbOKOnly, _
"MS-SQL Server Reported Error In - " & App.EXENAME
End Select
End Sub

```

在将当前示例的代码添加到 Vbsql1_ Message ()例程的定义体后, VB 应用程序将用一个 Windows 消息框显示 MS-SQL Server 在运行时向 DBLIB 报告的任何错误的错误代码及其描述。

技巧 317 使用 SqlColType()函数确定列的数据类型

正如在技巧 305 “使用 SqlData()从游标中将查询结果提取到应用程序”中学习过的, SqlData()函数允许将数据值从 DBLIB 缓冲区的列中提取到应用程序中的编程对象(如变量、结构域、窗体域或 MSFlexGrid 网格)。如果 DBLIB 缓冲区列包含 BINARY、VARBINARY 或 IMAGE 数据类型, SqlData()函数从结果列(DBLIB 缓冲区)返回由每个字节一个字符组成的二进制数据字符串。对所有的数据类型(数值型和字符/文本型)来说, SqlData()函数将返回一个可读字符的字符串。当 SqlData()函数以位串或字符串返回列数据值时, SqlColType()函数允许确定列数据的实际数据类型。

SqlColType()函数调用的句法如下:

```
iColType = SqlColType(nConnHandle, iColNo)
```

其中:

- Integer iColType 是表 15.2 中代表由 iColNo 的值指定的结果表列的数据类型的整型常量。

表 15.2 代表每个有效的 VB 或 SQL 数据类型的 Visual Basic (VB) 整型常量

SQL 列的数据类型	返回值	SQL 列的数据类型	返回值
binary (二进制型)	SQLBINARY	varbinary (变二进制型)	SQLBINARY
char (字符型)	SQLCHAR	varchar (变字符型)	SQLCHAR
datetime (日期型)	SQLDATETIME	smalldatetime (短日期型)	SQLDATETIME4
decimal (十进制型)	SQLDECIMAL	numeric (数值型)	SQLNUMERIC
float (浮点型)	SQLFLOAT8	real (实型)	SQLFLT4
image (图片型)	SQLIMAGE	text (文本型)	SQLTEXT
int (整型)	SQLINT4	smallint (短整型)	SQLINT2
		tinyint (微整型)	SQLINT1
money (货币型)	SQLMONEY	smallmoney (短货币型)	SQLMONEY4

- Long nConnHandle 是 SqlSendCmd()、SqlExec()或 SqlSend()用来向 MS-SQL Server 发

送供执行的 SQL 语句(或 set 语句集)的连接句柄(由 SqlOpenConnection()或 SqlOpen()函数返回)。

- Integer iColNoDBLIB 是函数要提取数据类型的缓冲区列号。DBLIB 缓冲区中的第一列是 1 号(而不是 0)。

所以,为了看出在 DBLIB 缓冲区中查询结果的第 3 列是否为 IMAGE 数据类型, Visual Basic 程序可执行这样的语句:

```
If SqlColType (nConnHandle, 3) = SQLIMAGE  
MsgBox "Use Image App. to display image in Col 3."  
End If
```

技巧 318 使用 SqlDatLen()函数确定储存在 DBLIB 缓冲区列中的数据字节数

对非数值数据类型(如 TEXT、CHAR 和 VARCHAR)来说,SqlDatLen()函数将返回 SqlData()函数能够从 DBLIB 缓冲区中的特定行提取的字符的个数。如果缓冲区数据类型包含数值型数据(如 SMALLINT、FLOAT 或 MONEY),则 SqlDatLen()将返回数据类型的最大可打印宽度(存储容量)而不是当前存储在 DBLIB 缓冲区中的数字的实际个数(要确定存储在数值型 DBLIB 缓冲区中的数字的个数,应使用 Visual Basic LEN()函数而不是调用 SqlDatLen()。

SqlDatLen()函数调用的句法如下:

```
iColLen = SqlDatLen(nConnHandle, iColNo)
```

其中:

- Integer iColLen 是调用 SqlData()函数来从非数值型 iColNo DBLIB 缓冲区列提取数据时,SqlData()函数将返回的字符的个数;或者如果列 iColNo 中的数据值是数值数据类型的一种,则是用来储存 iColNo 列的最大字节数。
- Long nConnHandle 是 SqlSendCmd()、SqlExec()或 SqlSend()用来向 MS-SQL Server 发送可供执行的 SQL 语句(或 set 语句集)的连接句柄(由 SqlOpenConnection()或 SqlOpen()函数返回)。
- Integer iColNo 是函数要确定其内容字节数的列号。DBLIB 缓冲区中的第一列是 1 号(而不是 0)。

例如,要在 nConnHandle 服务器连接的 DBLIB 缓冲区中的第一列显示字符数,可执行与以下类似的 Visual Basic (VB) 语句:

```
MsgBox "The number of letters in"" & _  
SqlData(nConnHandle,1) & """" & _  
" is " & SqlDatLen(nConnHandle, 1) & "."
```

作为对照,为了确定存储在数值列中的位数,可使用 VB 的 LEN()函数而不是 SqlDatLen()函数,如下所示:

```
MsgBox "The number of bytes used to store the decimal" & _  
" DISCOUNT value: " & SqlData(nConnHandle, 5) & _  
" is " & SqlDatLen(nConnHandle, 5) & "." & vbCrLf & _  
"The number of digits retrieved by SqlData() is " & _  
Len (SqlData(nConnHandle, 5)) & ".", vbOKOnly, _  
"SqlDatLen() Value vs. Len() Value for Numeric Data"
```

以上程序将显示如图 15.14 所示的消息框。



图 15.14 显示为由 SqlData()函数调用提取的数值型（十进制）数据调用 SqlDataLen()和 Len()函数的结果的 Windows 消息框

注意：如果要确定长度的 DBLIB 缓冲区列包含 NULL 值，则 SqlDataLen()函数将返回 0 作为 SqlData()能够从列提取的字节数。

技巧 319 在 Visual Basic 应用程序中给宿主变量指定 NULL 值

当 Visual Basic (VB) 变量定义不包括数据类型时，如：

```
DIM iHighQty
```

则 VB 将把变量确定为数据类型 variant——这意味着在程序执行的不同阶段变量可能有不同的数据类型。实际上，VB 应用程序能够用具有 variant 数据类型的变量保存标准数据类型的值或 3 种特殊值中的一种：Empty、Null 或 Error。

当第一次在 VB 程序中声明 variant 数据类型的变量（通过在变量声明中省略类型定义）时，系统给它指定 Empty 值。值 Empty 不同于 NULL。Empty 指 variant 类型的变量还没有指定值。只有通过执行下列语句指定 NULL 值的时候，VB 的 variant 型变量的值才会是 NULL：

```
iHighQty = Null
```

正如在技巧 305“使用 SqlData()函数从游标中将查询结果提取到应用程序中”中学习过的，SqlData()函数从连接的 DBLIB 缓冲区中的任何列以字符串作为返回值。如果 DBLIB 缓冲区列有 NULL 值，SqlData()函数将返回一个空的字符串（也就是返回的字符串将不包含任何字符），然而，该字符串不会是 NULL。所以，要将 NULL 值从 DBLIB 缓冲区列复制到数据类型为 variant 的宿主程序变量中，应用程序必须调用 SqlDataLen()函数检查储存在要提取其中内容的 DBLIB 缓冲区列中的数据的字节数。如果 SqlDataLen()函数返回值零（0），就把目标 variant 变量的值设为 NULL。

例如，下面的 VB 代码只要 nConnHandle 结果集中第 3 列的值为 NULL，就将显示文本 NULL。

```
If SqlDataLen(nConnHandle,3) = 0 Then
    iHighQty = NULL
End If
```

```
If IsNull(iHighQty) Then
    MsgBox "The HIGHQTY column for the discount type " & _
    SqlData(nConnHandle,3) & " is NULL."
End If
```

请注意 VB 像 SQL 一样，要求用特定函数 (IsNull()) 检查变量的值是否为 NULL。执行其中一个操作数是 NULL (If NULL = NULL) 的等式（不等式）测试永远不会是 TRUE，因为逻辑测试将总是被求值为 NULL。

VB 应用程序可将 NULL 值只赋给 variant 数据类型的变量。同样地，MSFlexGrid（其数

据类型被定义为 TEXT) 中的单元格不能保存 NULL。FlexGrid 单元格以说明单元格的底层 DBLIB 缓冲区列的值是 NULL。例如, 下面的 VB 代码:

```
For i = 1 To FlexGrid.Cols
    If SqlDataLen(nConnHandle, i) <> 0 Then
        FlexGrid.Text = SqlData(nConnHandle, i)
    Else
        FlexGrid.Text = "*** NULL ***"
    End If
Next i
```

当 SqlDataLen() 函数返回值 0 时, 把字符串 ** NULL ** 指定给 MSFlexGrid 中的所有单元格 (这说明单元格的底层 DBLIB 缓冲区列包含 NULL 值)。

技巧 320 使用 SqlSetOpt() 设置行缓冲区的大小以使用 SqlGetRow() 随机提取行

在技巧 306 “用 SqlNextRow() 函数在游标的行中移动” 中, 学习过如何用 SqlNextRow() 函数从 MS-SQL Server 保存查询结果集的 DBLIB 缓冲区中将行往前移 (一次一行)。默认情况下, DBLIB 允许只通过其缓冲区往前移。为了使缓冲区尽可能的小, 当程序通过调用 SqlNextRow() 函数移到下一行时, DBLIB 就放弃缓冲区中的当前行。毕竟, 由于应用程序不能在只能向前移的缓冲区中往后移, 所以没有理由在程序已经离开并移到下一行以后在缓冲区中保存这一行。

SqlSetOpt() 函数的 SQLBUFFER 选项告诉 DBLIB 在缓冲区中保持固定的行号, 直到应用程序调用 SqlClrBuff() 函数从缓冲区删除行。在调用带 SQLBUFFER 选项的 SqlSetOpt() 函数后, DBLIB 将允许应用程序调用 SqlGetRow() 函数。该函数允许程序移动到缓冲区中的任意行。读者将在技巧 321 “用 SqlGetRow() 函数在 DBLIB 查询结果缓冲区中选择当前行” 中学习 SqlGetRow() 函数的全部内容。目前, 要理解的重要事情是 SqlGetRow() 允许在 DBLIB 缓冲区中的任意行上建立当前行——不管目标行是在当前行的前面还是后面。在调用 SqlGetRow() 移动到 DBLIB 缓冲区中的特定行以后, 就可以调用 SqlNextRow() 函数通过缓冲区按顺序往前移, 或者再次调用 SqlGetRow() 函数移到另一行 (在当前行之前或之后)。

SqlSetOpt() 函数设置 DBLIB 要在查询结果缓冲区中维护的行数的句法如下:

```
nRetCode = SqlSetOpt(nConnHandle, SQLBUFFER, iBuffRowCt)
```

其中:

- Long nRetCode 是由 SqlSetOpt() 函数调用返回的整数结果。在调用函数以后, nRetCode 的值将是 SUCCEED (1) 或 FAIL (0)。
- Long nConnHandle 是 SqlSendCmd()、SqlExec() 或 SqlSend() 用来向 MS-SQL Server 发送供执行的查询的连接句柄 (由 SqlOpenConnection() 或 SqlOpen() 函数返回)。
- Integer iBuffRowCt 是 DBLIB 要在缓冲区中维护的查询结果的行数。iBuffRowCt 的值必须是 0 (表明没有缓冲区) 或在 2~32,767 之间。

所以, 要使 DBLIB 为查询结果集保持 10-行的缓冲区, 程序应当执行类似于下面的语句:

```
nRetCode = SqlSetOpt(nConnHandle, SQLBUFFER, 10)
```

就像前面已经提到的, 当应用程序调用带 SQLBUFFER 选项的 SqlSetOpt() 函数后, 它仍然可以调用 SqlNextRow() 函数从缓冲区中的下一行提取数据。而且, 应用程序也可以调用

SqlGetRow()函数任意次移到 DBLIB 缓冲区中的任意一行。例如, 下面的 Visual Basic 代码顺序地移过当前示例的 10-行 DBLIB 缓冲区的最初 3 行。

```
For iRow = 1 To 3
  nRetCode = SqlNextRow(nConnHandle)
  If (nRetCode <> NOMOREROWS) And
    (nRetCode <> BUFFULL) Then
    'Call SqlData() to retrieve specific data value from
    'columns in the "current" row of the query results set
  End If
Loop
```

如果程序准备调用 SqlNextRow()函数 4 次, DBLIB 将使缓冲区中的第 4 行成为当前行, 而且应用程序可以调用 SqlData()函数从 DBLIB 缓冲区的第 4 行(当前行)提取列值。然而, 如果程序准备以如下的方式调用 SqlGetRow()函数:

```
nRetCode = SqlGetRow(nConnHandle, 7)
```

则 DBLIB 将使用缓冲区中的第 7 行成为当前行, 而且随后对 SqlNextRow()函数的调用将使当前的行指针移到缓冲区的第 8 行。

技巧 321 使用 SqlGetRow()函数在 DBLIB 查询结果缓冲区中选择当前行

虽然 SqlNextRow()函数允许应用程序一次向前移过 DBLIB 缓冲区的一行, 但 SqlGetRow()函数却允许程序随机访问缓冲区中的任意行。在调用 SqlGetRow()前, 程序必须调用 SqlSetOpt()函数设置 DBLIB 要在其缓冲区中维护的查询结果的行数并调用 SqlNextRow()函数把查询结果的行从 DBMS 提取到 DBLIB 缓冲区中。

SqlGetRow()函数调用的句法如下:

```
nRetCode = SqlGetRow (nConnHandle, iRowNo)
```

其中:

- nRetCode (Long) 是由 SqlGetRow()函数调用返回的整数结果代码。当函数调用以后, nRetCode 的值将是下列值中的一个:
 - REGROW (-1), 如果缓冲区行 iRowNo 是标准的查询结果行。
 - COMPUTE 子句的 ID, 如果缓冲区行 iRowNo 是 COMPUTE 行。
 - FAIL (0)。
 - NOMOREROWS (-2), 如果行号 iRowNo 不是 DBLIB 缓冲区中的当前行。
- nConnHandle (Long) 是 SqlSendCmd()、SqlSend()或 SqlExec()用来将查询发送到 MS-SQL Server 供执行的句柄。
- iRowNo (Integer) 是函数要选取的查询结果的行号。例如, 如果 iRowNo 是 5, 则函数将尝试着从 DBLIB 缓冲区中的查询结果集查找第 5 行。

所以, 在进行下面的函数调用以后:

```
'Set the DBLIB buffer to 6 rows
nRetCode = SqlSetOpt(nConnHandle, SQLBUFFER, 6)
'Read 6 rows of data from the query results set into the
'DBLIB buffer
For i = 1 to 6
```

```
nRetCode = SqlNextRow(nConnHandle, i)
```

```
Next i
```

应用程序就可以如下地调用 `SqlGetRow()` 函数，从查询结果集选择第 4 行作为 DBLIB 的当前行：

```
nRetCode = SqlGetRow(nConnHandle, 4)
```

请记住，`SqlGetRow()` 函数仅仅告诉 DBLIB 把哪一行作为其当前行。要把数据值从当前行提取到它的变量或窗体上的域中，应用程序必须调用 `SqlData()` 函数。

可以调用 `SqlGetRow()` 函数移动到储存在 DBLIB 缓冲区中的查询结果的任意一行。然而，如果缓冲区是满的且要选取的行不是缓冲区中的当前行，就必须调用 `SqlClrBuf()` 例程（将在技巧 322 “使用 `SqlClrBuf()` 函数为 DBLIB 查询结果缓冲区中为附加行腾出空间” 中学习）来放弃缓冲区中的某些行，并随后调用 `SqlNextRow()` 例程重新填充缓冲区，直到要选取的查询结果集行处于 DBLIB 缓冲区中。

例如，如果正在使用一个 6 行的 DBLIB 缓冲区且已经执行一个生成 15 行结果的查询，可以通过执行下面的代码移到查询结果的第 12 行：

```
'Read 6 rows of query results into the DBLIB buffer
For i = 1 to 6
nRetCode = SqlNextRow(nConnHandle, i)
Next i
'Clear the DBLIB buffer to make room for more results
SqlClrBuf nConnHandle,6
'Read rows 7 - 12 of query results into the DBLIB buffer
For i = 1 to 6
nRetCode = SqlNextRow(nConnHandle, i)
Next i
'Select the 12th row of query results
nRetCode = SqlGetRow(nConnHandle,12)
```

请注意作为参数提供给 `SqlGetRow()` 函数的行号 (`iRowNo`) 指的是构成查询结果集的行集中行的位置，而不是它在储存在 DBLIB 缓冲区中的行内的位置。

技巧 322 使用 `SqlClrBuf()` 函数在 DBLIB 查询结果缓冲区中为附加行腾出空间

就像读者在技巧 321 “使用 `SqlGetRow()` 函数在 DBLIB 查询结果缓冲区中选择当前行” 的第一个示例中所看到的，`SqlSetOpt()` 函数的 `SQLBUFFER` 选项允许指定 DBLIB 要在其缓冲区中维护的查询结果的行数。例如，如果程序执行语句：

```
nRetCode = SqlSetOpt(nConnHandle, SQLBUFFER, 10)
```

则 DBLIB 将允许应用程序调用 `SqlNextRow()` 函数 10 次，从而把 10 行数据从查询结果集提取到 DBLIB 缓冲区中。在当前示例中，如果进行第 11 次调用，`SqlNextRow()` 函数将不能向 DBLIB 缓冲区添加另一行。作为结果，函数将不会提取一行查询结果并返回错误代码 `BUFFULL (-3)`。`SqlClrBuf()` 函数允许告诉 DBLIB 从其缓冲区中放弃一到多行以便为查询结果（这些行是可以随后调用 `SqlNextRow()` 函数提取的）的附加行腾出空间。

`SqlClrBuf()` 函数调用的句法如下：

```
SqlClrBuf nConnHandle, iRowCt
```

其中:

- `nConnHandle (Long)` 是 `SqlSendCmd()`、`SqlSend()` 或 `SqlExec()` 用来将查询发送到 MS-SQL Server 供执行的连接句柄 (由 `SqlOpenConnection()` 或 `SqlOpen()` 函数返回)。
- `iRowCt (Integer)` 是要从缓冲区中清除的查询结果的行数。如果 `iRowCt` 的值小于 1, DBLIB 将忽略 `SqlClrBuf()` 函数调用。另一方面, 如果 `iRowCt` 的值大于或等于缓冲区中的行数, DBLIB 将放弃缓冲区中的所有行。

DBLIB 按照先进先出的原则清除缓冲区中的行。所以, 如果有一个 10-行的缓冲区且已经调用 `SqlNextRow()` 函数 10 次用查询结果的开头 10 行填充它, 则执行语句:

```
SqlClrBuf nConnHandle, 5
```

将告诉 DBLIB 放弃已经添加到缓冲区中的开头 5 行 (通过调用 `SqlNextRow()` 函数)。

技巧 323 理解 MS-SQL Server 的 SELECT 语句中的 FOR BROWSE 子句

当 FOR BROWSE 子句被添加到 SELECT 语句时, 允许从另一个用户正在插入、更新或删除行的表中读取行。通常, MS-SQL Server 的锁定机制将防止从有未决的 (没有提交) 的 UPDATE、DELETE 或 INSERT 语句的表的页读取。

例如, 不带试图读取有未决的 (没有提交的) UPDATE 语句的行的 FOR BROWSE 子句的 SELECT 语句将“挂起”, 等待用户修改表的内容以执行 COMMIT 或 ROLLBACK 语句。如果用户在查询的超时间隔界满前提交 (或回滚) 未决的 UPDATE 语句, 则 SELECT 语句将返回它的结果表。否则, SELECT 语句将中止而不提取结果的任何行。FOR BROWSE 子句告诉 DBMS 允许查询执行而不需等待其他用户 COMMIT (提交) 或 ROLLBACK (回滚) 一个未决的 UPDATE、INSERT 或 DELETE 语句。

也就是说, 语句

```
SELECT * FROM employees
```

将等待其他用户 COMMIT (或 ROLLBACK) 未决的修改 EMPLOYEES 表的事务处理。而语句

```
SELECT * FROM employees FOR BROWSE
```

将在不等待的情况下生成结果表, 只要 EMPLOYEES 表同时有 TIMESTAMP 字段和 UNIQUE INDEX。

为了在 SELECT 语句中使用 FOR BROWSE 子句, 查询必须包含同时具有 TIMESTAMP 列和 UNIQUE INDEX 的单一表。而且, 如本例所示, FOR BROWSE 子句必须是 SELECT 语句中最后的子句。如果带 FOR BROWSE 子句的 SELECT 语句不满足这些要求, DBMS 将按没有 FOR BROWSE 子句的情况执行查询。

注意: 如果要查询的表没有 UNIQUE INDEX 或没有数据类型为 TIMESTAMP 的列, 可通过把会话的 TRANSACTION ISOLATION LEVEL 设置成 READ UNCOMMITTED 来获得与 SELECT 语句的 FOR BROWSE 子句近似的效果。请记住在被提交前读取更新或插入过的数据, 如果作出更改的用户通过执行 ROLLBACK 语句决定撤消改变, 将导致不可重复的异常读取。

技巧 324 理解 DBLIB 为什么不支持定位 UPDATE 和 DELETE 语句

与 ODBC 接口不同, DBLIB 不支持定位 UPDATE 和 DELETE 语句 (这是在技巧 292 “使

用 SQLSetPos 函数的 SQL_UPDATE 选项执行定位更新”和技巧 293 “使用 SQLSetPos 函数的 SQL_DELETE 选项执行定位删除”中学习的)。当应用程序调用 ODBC 驱动程序的 SQLFetch() 函数把一行数据从单个数据库表提取到游标中时, DBMS 就在查询结果的当前行与其底层表中的相应行保持直接的对应关系。使用这种对应关系, 应用程序可以构建与执行带 WHERE CURRENT OF <游标名>子句的 UPDATE 或 DELETE 语句。当提交给 DBMS 供执行时, 系统将更新 (UPDATE) 或删除 (DELETE) 表与在 WHERE CURRENT OF 子句中指定的游标中的当前行对应的行。当使用 DBLIB 接口时, 应用程序调用 SqlSendCmd()、SqlSend()或 SqlExec() 把 SELECT 语句发送给 DBMS 供执行。服务器发送查询结果集合 DBLIB, 而 DBLIB 使用其缓冲区储存数据。当程序调用使用 SqlNextRow()时, 该函数从 DBLIB 缓冲区而不是数据库本身中提取一行查询结果。作为结果, 在从 DBLIB 缓冲区提取到应用程序中的查询结果行与其列值被复制时的来源数据库表中的行之间没有由 DBMS 维护的对应关系。

不能把从 DBLIB 缓冲区提取的当前行与数据库表中的特定行联系起来, 使 DBLIB 不可能告诉 DBMS 执行定位 UPDATE 或 DELETE 语句。由于当用户被允许浏览一个查询结果集并且要 UPDATE (或 DELETE) 当前正在屏幕上显示的数据时缺乏定位更新或删除操作是真正的不足, 所以 DBLIB 提供了一组浏览模式的函数, 应用程序可用于针对 DBLIB 当前行借以派生出来的数据库行执行搜索更新 (UPDATE) 或删除 (DELETE)。

技巧 325 理解 DBLIB 浏览模式的函数

为了避免不能将当前 DBLIB 缓冲区行与其数据库表中的底层行关联起来的问题, DBLIB API 应当包括一组程序能用来创建伪定位更新能力的浏览模式函数 (将在技巧 326 “使用 SqlQual()函数为 DBLIB 浏览模式的 UPDATE 或 DELETE 语句生成 WHERE 子句”中学习)。DBLIB 浏览模式函数有:

- SqlTabCount(nConnHandle), 返回表的数目, 包括服务器工作表。用在使用作为此函数的惟一参数传递的连接句柄向服务器发送的 SELECT 语句中。
- SqlTabBrowse(nConnHandle,iTabNum), 如果由使用连接句柄 nConnHandle 向 DBMS 发送的查询所用的 iTabNum 表能够用 DBLIB 浏览模式例程更新, 则返回值 SUCCEED (1), 否则返回 FAIL (0)。DBLIB 查询中的第一个表是 table 1 (而不是 table 0)。所以在执行下列语句后, 如果查询中所用的第一个表能被 DBLIB 浏览模式例程更新, iCanBeUpdated 的值将是 1 (SUCCEED)。
`iCanBeUpdated = SqlTabBrowse(nConnHandle,1)`
- SqlTabName(nConnHandle,iTabNum), 返回在到使用连接句柄 nConnHandle 向 DBMS 发送的查询中使用的 iTabNum 表的名称。DBLIB 查询中的第一个表是 table 1 (而不是 table 0)。所以, 如下语句将发送把查询中所用的第二个表的名称置于字符串 sTableName 中:
`sTableName = SqlTabName(nConnHandle,2)`
- SqlColBrowse(nConnHandle,iColNum), 如果查询结果集中的 iColNum 列能用 DBLIB 浏览模式例程更新, 返回值 SUCCEED (1), 否则返回 FAIL (0)。DBLIB 结果集中的第一列是 column 1 (而不是 column 0)。
- SqlTabSource(nConnHandle,iColNum,iTabNum), 返回导出结果集列 iColNum 的表名。

并把表号置于 iTabNum 参数中。所以, 执行如下语句, 变量 sTableName 将包含查询中的第 4 列的源表的名称, 且 iTabNum 的值将包含它的表号。

```
sTableName = SqlTabSource(nConnHandle, 4, iTabNum)
```

- SqlColSource(nConnHandle, iColNum), 返回查询结果集中 iColNum 列的源表的列名。
- SqlQual(nConnHandle, iTabNum, sTabName), 返回可用于 UPDATE 或 DELETE 语句中的 WHERE 字符串, 从而可以改变或删除由 iTabNum 所代表的表号或由 sTabName 所代表的表名识别的数据库表查询结果的当前行。

技巧 326 使用 SqlQual()函数为 DBLIB 浏览模式的 UPDATE 或 DELETE 语句生成 WHERE 子句

DBLIB 能模拟定位 UPDATE 和 DELETE 语句, 只是因为 DBLIB 和 DBMS 实际上都不知道数据库表中的哪一行对应缓冲区中的当前行。然而, DBLIB 可以调用 SqlQual()函数生成可以在 UPDATE 或 DELETE 语句中执行搜索更新的 WHERE 子句, 从而在源数据库表中找出派生的 DBLIB 当前行。

SqlQual()函数的句如下:

```
sWhereClause = SqlQual(nConnHandle, iTabNum, sTabName)
```

其中:

- sWhereClause (String) 是由 SqlQual()函数返回的 WHERE 子句。WHERE 子句将包括对应于 DBLIB 中当前行的表号为 iTabNum 或表名为 sTabName 的表中 UNIQUE INDEX 列和 TIMESTAMP 列的列名和值。
- nConnHandle (Long) 是 SqlSendCmd()、SqlSend()或 SqlExec()用来将查询发送到 MS-SQL Server 供执行的连接句柄 (由 SqlOpenConnection()或 SqlOpen()函数返回)。
- iTabNum (Integer) 是 SELECT 语句的 FROM 子句中将成为浏览模式 UPDATE 或 DELETE 语句的目标的表号。表在 FROM 子句中是由 1 开始从左边编号的) 如果 iTabNum 是 -1, SqlQual()函数将用 sTabName 中的字符串识别表。
- sTabName (String) 是要被 DBLIB 浏览模式 UPDATE 或 DELETE 语句更新的表的名称。如果 sTabName 是一个空字符串, SqlQual()函数将用 iTabNum 的值识别表。

所以, 如果这样调用函数 SqlGetRow():

```
SqlGetRow(nQueryConnHandle, 5)
```

来选择由查询返回的结果的第 5 行并随后如下调用 SqlQual()函数:

```
sWhereClause = SqlQual(nQueryConnHandle, -1, "employees422")
```

字符串变量 sWhereClause 将包含类似于下面的 WHERE 子句:

```
where (emp_ID=5) and tsequal(timestamp, 0x000000000000006fc)
```

可以随后将该 WHERE 子句添加到 DELETE 语句 (正如读者将在技巧 327 “执行 DBLIB 浏览模式的 DELETE 语句”中学习的) 或 UPDATE 语句 (正如读者将在技巧 328 “执行 DBLIB 浏览模式的 UPDATE 语句”中学习的), 从而在数据库中删除或更新 DBLIB 的当前行。

技巧 327 执行 DBLIB 浏览模式的 DELETE 语句

在尝试 DBLIB 浏览模式的 DELETE 语句之前, 请牢记要删除的当前 DBLIB 缓冲区行的源表必须同时有 TIMESTAMP 和 UNIQUE INDEX 数据类型的列。在选定符合条件的表以后,

打开两个与 MS-SQL Server 的连接——一个是要通过它将查询发送给 DBMS (并提取其结果集) 而第二个用于发送浏览模式的 DELETE 语句。例如, 如下 Visual Basic (VB) 代码:

```
Dim sDelStmt As String
Dim nRetCode As Long
Dim nQueryConnHandle As Long
Dim nDelConnHandle As Long
nQueryConnHandle=SqlOpenConnection("NVBizNet2","konrad",_
"king","ws-query",App.EXENAME)
nDelConnHandle=SqlOpenConnection("NVBizNet2","konrad",_
"king","ws-delete",App.EXENAME)
```

将建立两个与 MS-SQL Server NVBizNet2 的连接。

接着, 选择包含带有要删除的行的表的数据库。然后, 用查询连接句柄向 DBMS 发送带 FOR BROWSE 子句的 SELECT 语句:

```
nRetCode = SqlCmd(nQueryConnHandle," USE SQLTips")
nRetCode = SqlCmd(nQueryConnHandle,
" SELECT * FROM employees423 FOR BROWSE")
nRetCode = SqlSend(nQueryConnHandle)
```

在 DBMS 执行带 FOR BROWSE 的 SELECT 语句以后, 通过执行与下面类似的 VB 代码提取查询结果集:

```
nRetCode = SqlOk(nQueryConnHandle)
nRetCode = SqlResults(nQueryConnHandle) 'Get USE results
nRetCode = SqlResults(nQueryConnHandle) 'Get SELECT results
nRetCode = SqlSetOpt(nQueryConnHandle, SQLBUFFER, 100)
Do Until NOMOREROWS = SqlNextRow(nQueryConnHandle)
Loop
```

现在, 假设要删除对应于查询结果集中第 10 行的 EMPLOYEES423 表中的行。首先, 用 SqlGetRow() 函数选择查询的第 10 行结果作为 DBLIB 的当前行:

```
nRetCode = SqlGetRow(nQueryConnHandle,10)
```

然后调用 SqlQual() 函数创建搜索的 DELETE 语句可以用来从 EMPLOYEES423 表查找并删除行的 WHERE 子句:

```
sDelStmt = "DELETE FROM employees423 " & _
          SqlQual(nQueryConnHandle,-1,"employees423")
```

最后, 用 nDelConnHandle 中的 DELETE 连接句柄将 DELETE 语句发送到 MS-SQL Server 供执行:

```
nRetCode = SqlSendCmd(nDelConnHandle,sDelStmt)
```

技巧 328 执行 DBLIB 浏览模式的 UPDATE 语句

与 DBLIB 浏览模式 DELETE 语句的情况相似, 只可以在同时有 TIMESTAMP 和 UNIQUE INDEX 数据类型的列的表上执行 DBLIB 浏览模式 UPDATE 语句。要执行 DBLIB 浏览模式 UPDATE 语句, 首先要打开两个与 MS-SQL Server 的连接——一个是通过它发送查询而第二个由用来发送 UPDATE 语句。例如, Visual Basic (VB) 代码:

```
Dim sUpdtStmt As String
Dim nRetCode As Long
```

```
Dim nQueryConnHandle As Long
Dim nUpdtConnHandle As Long
nQueryConnHandle=SqlOpenConnection("NVBizNet2","konrad",_
"king","ws-query",App.EXEName)
nUpdtConnHandle=SqlOpenConnection("NVBizNet2","konrad",_
"king","ws-update",App.EXEName)
```

将建立两个与 MS-SQL Server NVBizNet2 的连接。

接着，选择含有要更新的行的表的数据库。然后用带查询连接句柄的连接向 DBMS 发送带 FOR BROWSE 子句的 SELECT 语句：

```
nRetCode = SqlCmd(nQueryConnHandle," USE SQLTips")
nRetCode = SqlCmd(nQueryConnHandle,
" SELECT * FROM employees424 FOR BROWSE")
nRetCode = SqlSend(nQueryConnHandle)
```

在向 DBMS 发送供执行的带 FOR BROWSE 的 SELECT 语句以后，通过执行与下面类似的 VB 代码提取查询结果集：

```
nRetCode = SqlOk(nQueryConnHandle)
nRetCode = SqlResults(nQueryConnHandle) 'Get USE results
nRetCode = SqlResults(nQueryConnHandle) 'Get SELECT results
nRetCode = SqlSetOpt(nQueryConnHandle, SQLBUFFER, 100)
Do Until NOMOREROWS = SqlNextRow(nQueryConnHandle)
Loop
```

现在假设要更新对应于查询结果缓冲区中第 7 行的 EMPLOYEES423 表中的行。首先，用 SqlGetRow()函数选择查询的第 7 行结果作为 DBLIB 缓冲区中的当前行：

```
nRetCode = SqlGetRow(nQueryConnHandle,7)
```

然后调用 SqlQual()函数创建搜索的 UPDATE 语句能用来查找底层行(在 EMPLOYEES423 表中)的 WHERE 子句。该底层行是 DBLIB 中当前行的数据源。例如，要在与 DBLIB 的当前行对应的底层行中将 POSITION 列设置为 NULL，请执行与下面类似的 VB 语句：

```
sUpdtStmt = "UPDATE employees423 SET position = NULL " & _
SqlQual(nQueryConnHandle,-1,"employees423")
nRetCode = SqlSendCmd(nUpdtConnHandle,sUpdtStmt)
```

技巧 329 用 DBLIB API 执行动态 SQL 查询

虽然技巧 322~技巧 328 中的工程中所使用的 SELECT 语句在每个 Visual Basic (VB) 示例程序中都被硬性编码了，DBLIB 函数也允许编写能让用户在运行时程式化查询的应用程序。这种动态 SQL 查询能力用 MSFlexGri 控件工作得特别好，它允许及时定义其列名并设置其列与行的计数。

例如，如果设计带如图 15.15 所示的文本域(名为 Text1)的 VB 窗体，可以如下这样调用 SqlSendCmd()函数：

```
nRetCode = SqlSendCmd(nQueryConnHandle,Form1.Text1.Text)
```

把 Text1 的内容放进 nQueryConnHandle 命令缓冲区并把缓冲区的内容发送给 DBMS 供执行。

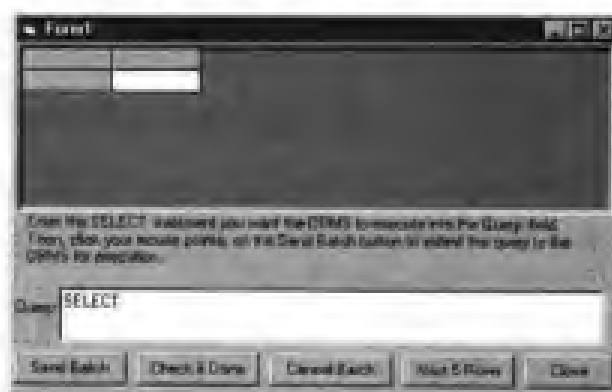


图 15.15 带有在运行时接受用户的动态 SQL 语句的文本域的 Visual Basic 窗体

假设 Form1 上的 Text1 域包含有效的 SELECT 语句，可用来查询 VB 程序的连接拥有 SELECT 访问权的表，VB 代码如下：

```
Dim i As Integer
FlexGrid.Cols = SqlNumCols(nQueryConnHandle)
FlexGrid.Rows = 1
FlexGrid.Row = 0
FlexGrid.Col = 0
For i = 1 To FlexGrid.Cols
    FlexGrid.Text = SqlColName(nQueryConnHandle, i)
    If FlexGrid.Col < FlexGrid.Cols - 1 Then
        FlexGrid.Col = FlexGrid.Col + 1
    End If
Next i
```

将从查询结果表中提取列的计数及其名称。

如果随后把用前面的 VB 代码设置的带有列名和列数的 MSFlexGrid 连同查询连接句柄一起传递给如下 VB 例程：

```
Sub GetResultsSet (nConnHandle As Long, _
    FlexGrid As MSFlexGrid)
    Dim iCol As Integer
    Dim nRetCode As Long
    Do Until NOMOREROWS = SqlNextRow(nConnHandle)
        FlexGrid.Col = 0
        FlexGrid.Rows = FlexGrid.Rows + 1
        FlexGrid.Row = FlexGrid.Rows - 1
        For iCol = 1 To FlexGrid.Cols
            If SqlDatLen(nConnHandle, iCol) <> 0 Then
                FlexGrid.Text = SqlData(nConnHandle, iCol)
            Else
                FlexGrid.Text = "*** NULL ***"
            End If
            If FlexGrid.Col < (FlexGrid.Cols - 1) Then
                FlexGrid.Col = FlexGrid.Col + 1
            End If
        Next iCol
    End Do
```

```
Loop  
End Sub
```

这样就有了允许用户查询应用程序连接拥有适当访问权限的任何表的 VB 应用程序。而且，程序的 MSFlexGrid 对象将以正确的列名（即使列已被选定）、列数和行数显示查询结果，而具体的显示项目可能随用户输入到 Text1 域中并发送到 MS-SQL Server 供执行的每个新的动态 SQL 查询而定。

第 16 章 通过游标提取和维护数据

技巧 330 理解游标的目的

与 Windows 或 DOS 的“光标”(cursors)(它们显示在文档、域或 DOS 命令行中文本的当前插入点)不同,SQL 的游标是一种临时的数据库对象,既可用于放置储存在系统永久表中的数据行的副本,也指向储存在系统永久表中的数据行的指针。游标给出了在逐行的基础上而不是一次处理整个结果集为基础的操作表中数据的方法。

例如,如果想要给顾客在 12/01/00~12/31/00 期间所下的定单以总价 1%的折扣,可以执行与下面类似的 UPDATE 语句:

```
UPDATE customers SET rebate =  
(SELECT SUM(order_total) FROM orders  
WHERE order_date BETWEEN '12/01/2000' AND '12/31/2000'  
GROUP BY cust_no  
HAVING cust_no = cust_ID) * .01
```

当前示例中的子查询生成每个顾客在 2000 年 12 月份购货的总量的清单的结果集。UPDATE 语句随即把它作为一个组将 1%的折扣应用到该定单并把每个顾客应得的返还款放到顾客在 CUSTOMERS 表行中的 REBATE 列。所以,当把相同的百分比应用于组中定单的每一行时,SQL 一次处理一个结果集的数据(此处是指每个顾客在 2000 年 12 月份的定单)的标准方法是一种计算折扣的有效方法。

现在,假设要根据顾客在相同的时间内购买量的多少,给范围为 1%~2%的折扣。顾客在购买第一个\$1,000.00 时,会得到 1%的折扣,对下一个\$1,000.00 价值的定单折扣为 1.5%,而且在顾客累积的定单总价达到\$2,000.00 以后额外购买的部份折扣为 2%。虽然仍然需要处理每个顾客在 2000 年 12 月份作出的定单组,但不能再把相同的折扣应用于结果集中的每个定单行。在这种情况下,游标就有用了,因为它允许以一次处理一个定单的方式处理每个顾客的定单。通过在从一个定单移到下一个定单时跟踪累积的购买量,可以在以前的定单总价从折扣表的一级移到下一级时应用不同(较高)的折扣率到所下的定单。

读者将在技巧 331~技巧 345 中学习如何创建并使用游标处理数据。现在,要理解的重要事情是游标是允许以一次一行的方式处理底层表中的数据的临时数据库对象。当需要在相同的数据集上执行多个动作时,如果用游标工作则用游标保存数据比多次执行相同的操作效率更高——每个需要从结果集获取数据的动作要执行一次。

技巧 331 使用 DECLARE CURSOR 语句定义游标

SQL DECLARE CURSOR 语句不仅允许指定用于生成 DBMS 储存于游标中的结果集(行集)的查询,也允许指定用户(或应用程序)使用游标时采取的动作。当通过执行如下 DECLARE CURSOR 语句声明游标时:

```
DECLARE cur_payroll_work CURSOR
```

```
FOR SELECT emp_num, dept, hourly_rate, ot_rate,  
monthly_salary, time_in, time_out, project,  
tcard_hourly_rate, tcard_labor_cost  
FROM timecards, employees  
WHERE timecards.emp_ID = employees.emp_num
```

SQL 服务器确认游标的查询，确保其句法正确以及 SELECT 语句的 FROM 子句中列出的表或视图在数据库中确实存在。

DECLARE CURSOR 语句的句法如下：

```
DECLARE <cursor name> [INSENSITIVE][SCROLL] CURSOR  
FOR <SELECT statement>  
[FOR {READ ONLY|UPDATE[OF <column list>]}]
```

其中：

- **INSENSITIVE** 告诉 DBMS 制作查询结果集数据的临时副本（而不是使用指针引用永久数据库表中真实数据行中的列）。如果正在使用（UPDATE 和 DELETE）INSENSITIVE 游标，对底层表作出的任何改动都不会反映到游标的数据中。而且，INSENSITIVE 游标是 READ ONLY 的（即只读的），这意味着不能修改其内容或用 INSENSITIVE 游标修改其底层表的内容。如果 INSENSITIVE 选项在 DECLARE CURSOR 语句中被省略，则 DBMS 将创建能反映对永久数据库表中底层行所作的任何改动都对 UPDATE 和 DELETE 敏感的游标。
- **SCROLL** 指定游标支持通过使用任意 FETCH 选项（FIRST、LAST、PRIOR、NEXT、RELATIVE 和 ABSOLUTE）选取它的任意行作为当前行。如果 SCROLL 选项在 DECLARE CURSOR 语句中被省略，则游标将只支持经过它的行向前移动单个行（也就是，它将只支持移过游标的 FETCH NEXT 选项）。
- **READ ONLY** 防止正在使用游标的用户（或应用程序）通过更新数据或删除行改变游标的内容。因此，不能用 READ ONLY 游标修改游标的底层表中的数据。如果 READ ONLY 选项在 DECLARE CURSOR 语句中被省略，则 DBMS 将创建能用来修改其基表的游标。
- **UPDATE** 就是告诉 DBMS 创建可更新游标且（可选的）列出值能被更新的游标列的 UPDATE 子句。如果 UPDATE 子句中被列入了任意列，则只有被列入的列才能被更新。另一方面，如果 DECLARE CURSOR 语句只指定了 UPDATE 选项（没有列的清单），则游标将允许更新它的任何或所有列。

所以，在本技巧开头的 DECLARE CURSOR 语句定义（声明）一个名为 PAYROLL_WORK 的“只能往前”的游标，它显示 TIMECARDS 表中的所有列与行以及 EMPLOYEES 表中行的一些列值。

技巧 332 使用 OPEN 语句创建游标

当执行 DECLARE CURSOR 语句时，DBMS 只确认游标的 SELECT 语句。相反，当执行 OPEN 语句时，DBMS 不仅创建游标，还通过执行游标的 SELECT 语句用数据填充游标。如果 DECLARE CURSOR 语句包括 INSENSITIVE 选项，OPEN 语句将导致 DBMS 创建保存查询结果的临时表。否则，DBMS 将把游标查询结果集中每行的行 ID 放进由游标管理的临时工

作空间。

（游标）OPEN 语句的句法如下：

```
OPEN <cursor name>
```

同样，为创建和填充在技巧 331 “用 DECLARE CURSOR 语句定义游标”中开头所创建的游标，可执行如下 OPEN 语句：

```
OPEN cur_payroll_work
```

因为在当前示例中对游标的声明不包括 INSENSITIVE 选项，DBMS 将用底层表（在游标查询的 FROM 子句中列出）中满足游标查询的 WHERE 子句中搜索条件的每一行的行 ID 填充游标的存储结构（如果 DECLARE CURSOR 语句已经包括 INSENSITIVE 选项，则 DBMS 将创建一个表并用游标的底层表中相应行的数据的副本填充）。

在完成使用游标的工作后，就要 CLOSE（关闭）游标以释放游标所占用的系统资源（如硬盘空间和服务器内存）并释放任何对当前被游标使用的行（或内存页）的锁定。CLOSE（游标）语句的句法如下：

```
CLOSE <游标名>
```

从概念上讲，执行此语句将删除 DBMS 用以下语句打开游标时创建的查询结果表：

```
OPEN <游标名>
```

与 DBLIB 缓冲区不同，不必在关闭游标前 FETCH（获取）并处理其中的所有行。只要记住关闭游标时剩下的没有处理完的行对任何 DBMS 用户、外部应用程序或存储过程都不再可用。

技巧 333 使用 ORDER BY 子句改变游标中行的顺序

如果需要按特别的顺序处理游标数据，可通过在 DECLARE CURSOR 语句中把 ORDER BY 子句添加到查询来将游标数据排序。例如，如果正在对工资单系统进行工作且有包含周考勤卡数据的游标，最好将考勤卡在雇员号内按日期排序。通过将相同雇员的所有考勤卡按日期顺序分组，可以使一个雇员的数据经过游标中的各行将在此期间内雇员小时数进行合计并计算雇员每张考勤卡的报酬率。只要该周总的工时数超过 40 小时，就知道在计算雇员的小时工资时要改成按超时率计算雇员其余的考勤卡。然后，只要 FETCH（获取）一个与刚才处理的雇员有不同雇员号的考勤卡，就知道要在新的考勤卡中重置总的小时数，因为已经移到下一个雇员。

用来设置游标中行的顺序的 ORDER BY 子句的句法与正常的 ORDER BY 子句相同：

```
ORDER BY <column name> [ASC | DESC]
[,...<last column name> [ASC | DESC]]
```

所以，为了将 CUR_PAYROLL_WORK 游标（在技巧 331 “用 DECLARE CURSOR 语句定义游标”中定义的）内的 TIMECARDS 在雇员号内按日期和时间的升序进行排序，可把游标的定义改成：

```
DECLARE cur_payroll_work CURSOR
FOR SELECT emp_num, dept, hourly_rate, ot_rate,
monthly_salary, time_in, time_out, project,
tcard_hourly_rate, tcard_labor_cost
FROM timecards, employees
WHERE timecards.emp_ID = employees.emp_num
ORDER BY emp_ID, time_in, time_out
```


由于默认的排序顺序是升序，不必在任何在 ORDER BY 子句中列出的列后都包括 ASC 选项来按升序对查询结果表进行排序。然而，如果要将任何在列清单中给出了名称的列按降序排序，就必须在列名后包括 DESC 选项。

注意：与非游标 SELECT 语句中的 ORDER BY 子句不同，只有在查询的 SELECT 子句中列出的供显示的列才能作为 ORDER BY 子句中的列出现（在非游标的 SELECT 语句中，表中任何在查询的 FROM 子句中列出的列都可能出现在 ORDER BY 子句中——即使列没有在语句的 SELECT 子句中列出[供显示]）。

技巧 334 在游标中包含计算好的值作为列

在游标 SELECT 语句的 FROM 子句中列出的除了表中的列和视图外，游标还可以包含计算好的列。例如，假设公司中的一些雇员按月支付薪水而不是按小时计酬。然而，为了计算公司每个项目的劳动力消耗，付月薪的雇员提交可显示在每个项目中的工作时间的考勤表，并且根据下面的公式计算小时报酬率：

$$(\text{<monthly salary>} * 12) / (40 \text{ hours} * 52 \text{ weeks})$$

如果付月薪的雇员的 HOURLY_RATE 列值是 NULL 且计时计酬的雇员 MONTHLY_SALARY 列值是 NULL，就可以用 COALESCE 函数通过把 DECLARE CURSOR 语句改成下面的样子向 CUR_PAYROLL_WORK 游标添加计算好的 HRLY_PAYRATE 列：

```
DECLARE cur_payroll_work CURSOR
FOR SELECT emp_num, dept, ot_rate, monthly_salary,
time_in, time_out, project,
tcard_hourly_rate, tcard_labor_cost,
COALESCE(hourly_rate,
(monthly_salary * 12) / 2080) hrly_rate
(CONVERT(REAL, (time_out - time_in), 8) * 24)
hours_worked
FROM timecards, employees
WHERE timecards.emp_ID = employees.emp_num
ORDER BY emp_ID, time_in, time_out
```

除了 HRLY_RATE 列，当前示例中的游标还包括计算好的列 HOURS_WORKED，以避免在处理游标数据时不得不计算每个考勤卡上工作的时间数。

技巧 335 使用 FOR UPDATE 子句指定游标可修改底层表的哪些列

默认情况下，可更新的游标允许 UPDATE（更新）所有在用于定义游标的查询的 SELECT 子句中列出的列中的值。然而，游标允许用它来一次 DELETE（删除）其底层表的一行。如果要防止用户能用游标修改底层表中的列值或从中删除行，需将 READ ONLY 子句添加到 DECLARE CURSOR 中，如下所示：

```
DECLARE <cursor name> CURSOR
FOR <SELECT statement> READ ONLY
```

相反，如果在 DECLARE CURSOR 语句中既省略 READ ONLY 子句也省略 FOR UPDATE 子句，或包含一个 FOR UPDATE 子句而没有列的清单，则用户可用游标 UPDATE（更新）在游标的 SELECT 语句中列出任何列的值。然后，通过执行带 WHERE CURRENT OF <游标名> 子句的 UPDATE 或 DELETE 语句，用户能用游标对底层表中的列作出改动或从中删除行（读

者将在技巧 338 “理解基于游标的定位 DELETE 语句”和技巧 339 “理解基于游标的定位 UPDATE 语句”中学习定位更新和删除)。

所以, DECLARE CURSOR 语句:

```
DECLARE <cursor name> CURSOR
FOR <SELECT statement>
```

和

```
DECLARE <cursor name> CURSOR
FOR <SELECT statement> FOR UPDATE
```

将允许用户 UPDATE (更新) 在游标的 SELECT 语句的 SELECT 子句中列出的任何列。如要限制用户只能更新游标的某些列中的值, 需在 DECLARE CURSOR 语句的 FOR UPDATE 子句中列出符合更新条件的列。

例如, 在 DECLARE CURSOR 语句的 FOR UPDATE 子句中列出两列

```
DECLARE cur_payroll_work CURSOR
FOR SELECT emp_num, dept, ot_rate, monthly_salary,
time_in, time_out, project,
tcard_hourly_rate, tcard_labor_cost,
COALESCE(hourly_rate,
(monthly_salary * 12) / 2080) hrly_rate
(CONVERT(REAL, (time_out - time_in), 8) * 24)
hours_worked
FROM timecards, employees
WHERE timecards.emp_ID = employees.emp_num
FOR UPDATE OF tcard_hourly_rate, tcard_labor_cost
```

将只允许用户使用 UPDATE (更新) 游标和 TIMECARDS 表的 TCARD_HOURLY_RATE 和 TCARD_LABOR_COST 列中的值。

注意: 如果将 ORDER BY 子句添加到游标的 SELECT 语句, DBMS 将把游标限制为 READ ONLY (只读的)。同样, 不能在相同的游标 SELECT 语句中同时有 ORDER BY 子句和 FOR UPDATE 子句。

技巧 336 使用 FETCH 语句从游标中的行提取列值

FETCH 语句允许从游标中的下一行提取或显示数据值。当执行 OPEN (游标) 语句时, DBMS 打开 OPEN 关键词后面跟着的游标名的游标, 并用游标的 SELECT 语句的结果填充游标, 把游标的当前行指针放在游标的第一行之前。所以, 当在打开游标后执行的第一个 FETCH 语句从游标中的“下一”行提取数据时, 它将返回游标第一行中的数据值。

FETCH 语句的句法如下:

```
FETCH [NEXT|PRIOR|FIRST|LAST|
{ABSOLUTE <row number>}|{RELATIVE <row number>}]
FROM <cursor name>
[INTO <variable name>[,...<last variable name>]]
```

所以为了显示由下面的语句声明的 CU_AUTH 游标中前两位作者的 ID 和姓名:

```
USE pubs
DECLARE cur_authors CURSOR
```

```
FOR SELECT au_ID, au_fname, au_lname FROM authors
```

可执行下面的 Transact-SQL 代码：

```
DECLARE @au_ID CHAR(11),  
        @au_fname VARCHAR(30), @au_lname VARCHAR(30)  
OPEN cur_authors  
FETCH cur_authors INTO @au_ID, @au_fname, @au_lname  
PRINT 'ID: ' + @au_ID + ' Name: ' + @au_fname + ' ' + @au_lname
```

这将产生与下面类似的结果：

```
ID: 172-32-1176 Name: Johnson White  
ID: 213-46-8915 Name: Marjorie Green
```

每次执行 FETCH 语句时，DBMS 都移到游标中的下一行并把游标中的列值获取（复制）到语句的 INTO 子句中列出的临时变量中。FETCH 语句从左至右地操作游标列及其 INTO 子句中的变量，把游标列中的值复制到变量表中的相应变量中。同样，FETCH 语句的 INTO 子句中列出的变量必须与游标的 SELECT 语句中的列清单在个数和数据类型上都匹配（如果 FETCH 语句没有 INTO 子句，DBMS 将把游标当前行中所有列的值显示在屏幕上）。

请注意，如果 FETCH 语句不包含行选项（ABSOLUTE、RELATIVE、FIRST、LAST）或移动方向（NEXT、PRIOR），DBMS 会将语句执行成 FETCH NEXT。

技巧 337 把游标的当前行指针预先定向到从当前行获取列值

正如读者在技巧 336 “使用 FETCH 语句从游标中的行提取列值”中所看到的，如果执行不包含移过游标方向的 FETCH 语句，如：

```
FETCH FROM <cursor name> INTO <variable list>
```

则 DBMS 会假定想要执行 FETCH NEXT 语句。同样，系统将游标的行指针从当前位置前移到游标中的一行并使 FETCH 语句将列值从该行提取到语句的 INTO 变量中——或者如果没有 INTO 子句的话则将它们显示在屏幕上。如果在声明游标中不包含 SCROLL 选项，则使用 FETCH NEXT 语句时，DBMS 将只允许一次向前移过游标一行。

仅当使用 SCROLL 游标时，才能使用 FETCH 语句的行定位选项（除了 NEXT 以外）。为了声明 SCROLL 游标，请在游标声明中包括关键词 SCROLL，如下所示：

```
DECLARE <cursor name> SCROLL CURSOR  
FOR <SELECT statement>
```

当使用 SCROLL 游标时，可以使用任何 FETCH 语句的游标行指针的定位选项预先移动到游标中的一个特定行来将其列值提取到临时变量（或屏幕）中。FETCH 语句的定位选项有：

- NEXT 向前移动一行。
- PRIOR 向后移动一行。
- FIRST 移动到结果集中的第一行。
- LAST 移动到结果集中的最后一行。
- ABSOLUTE *n* 移动到结果集中的第 *n* 行。如果 *n* 是正值，DBMS 从游标的顶部向前移动到第 *n* 行。如果 *n* 是负值，则 DBMS 从结果集的底部移到第 *n* 行。
- RELATIVE *n* 从行指针的当前位置移动 *n* 行。如果 *n* 是正值，DBMS 将行指针向前移动 *n* 行。如果 *n* 是负值，则 DBMS 将行指针向后（向游标的顶部）移动 *n* 行。

所以，如果正在使用如下声明的游标：

```
DECLARE cur_authors SCROLL CURSOR
FOR SELECT au_fname, au_lname FROM authors
```

则执行 FETCH 语句:

```
FETCH RELATIVE 1 FROM cur_authors INTO @au_fname, @au_lname
```

就有与如下语句相同的效果:

```
FETCH NEXT FROM cur_authors INTO @au_fname, @au_lname
```

在这两个语句中都在从游标的（新）当前行提取列值前把游标的当前行指针前移一行。

相似地, FETCH 语句:

```
FETCH RELATIVE -1 FROM cur_authors
INTO @au_fname, @au_lname
```

则有与下面语句相同的效果:

```
FETCH PRIOR FROM cur_authors INTO @au_fname, @au_lname
```

在这两个语句中都把游标的当前行指针从当前行向后（向游标中的第一行）移动一行。

现在, 假设当前行指针在一个 20-行的游标的第 5 行。以下 FETCH 语句:

```
FETCH RELATIVE 10 FROM cur_authors
INTO @au_fname, @au_lname
```

将把当前行指针向前移动 10 行到第 15 行, 而以下 FETCH 语句:

```
FETCH ABSOLUTE 10 FROM cur_authors
INTO @au_fname, @au_lname
```

则将会把当前行指针移到游标的第 10 行。相似地, 以下 FETCH 语句:

```
FETCH RELATIVE -3 FROM cur_authors
INTO @au_fname, @au_lname
```

将把游标的当前行指针从当前的第 5 行移到游标的第二行（这里是向上 3 行[相对于游标的顶部]），而以下 FETCH 语句:

```
FETCH ABSOLUTE -3 FROM cur_authors
INTO @au_fname, @au_lname
```

将移动游标的当前行指针到游标的第 17 行, 这是从游标底部（第 20 行）向后 3 行。

技巧 338 理解基于游标的定位 DELETE 语句

如果游标是可更新的（也就是, 用来声明游标的 DECLARE CURSOR 语句不包含 READ ONLY 子句），就可以用游标从游标数据的源表中 DELETE（删除）行。基于游标的 DELETE 语句称为定位 DELETE 语句, 因为这告诉 DBMS 从基于游标行指针的当前位置的表中删除一行。

标准（搜索的）DELETE 语句中的 WHERE 子句描述了基于一列或多列中的需要删除值的行。例如, 搜索的 DELETE 语句:

```
DELETE FROM orders
WHERE date_shipped = NULL AND order_date < GetDate() - 30
```

告诉 DBMS 搜索并 DELETE（删除）任何 DATE_SHIPPED 列值是 NULL 以及 ORDER_DATE 列值比当前日期早 30 天以上的行。

另一方面, 定位 DELETE 语句告诉服务器 DELETE（删除）底层表中与游标的当前行相关联的行, 如下面语句所示:

```
DELETE FROM <table name> WHERE CURRENT OF <cursor name>
```

所以,要删除游标中的第 3 行并从游标的底层表 DELETE(删除)行,应当首先执行 FETCH 语句使第 3 行成为游标的当前行,并且随后执行定位 DELETE 语句。例如,要从游标 CU_TIMECARDS 删除第 3 行以及游标中第 3 行在源表中的相应行,应当执行与下类似的语句批处理:

```
OPEN cur_timecards CURSOR
FETCH FROM cur_timecards
FETCH FROM cur_timecards
FETCH FROM cur_timecards
DELETE FROM timecards WHERE CURRENT OF cur_timecards
```

注意:对于具体的 DBMS,一定要检查各种 DECLARE CURSOR 子句对游标的 READ ONLY 状态上的效果。例如,MS-SQL Server 强制把 SCROLL 游标设为 READ ONLY。然而,MS-SQL Server 允许向 DECLARE 游标语句添加 DYNAMIC 子句来创建可滚动且可更新的游标。

技巧 339 理解基于游标的定位 UPDATE 语句

除了执行定位 UPDATE 语句,也可用可更新游标执行定位更新。定位 UPDATE 语句允许设置底层表的是游标中当前行的来源的行中的列值,所以,定位 UPDATE 语句类似于定位 DELETE 语句,因为在其 WHERE 子句中的搜索条件是基于游标的当前行指针的位置而不是行的一列或多列中的值。

标准(搜索的)UPDATE 语句中的 WHERE 子句描述基于一列或多列中的值要修改的行。例如,以下搜索的 UPDATE 语句:

```
UPDATE employees SET ot_payrate = 1.5
WHERE monthly_salary IS NULL
```

告诉 DBMS 搜索 EMPLOYEES 表中 MONTHLY_SALARY 列的值是 NULL 的行,并随后把这些行中 OT_PAYRATE 列的值设为 1.5。

与此同时,定位更新使用如下句法:

```
UPDATE <table name> SET <column name> = <value>
[,...<last column name> = <last value>]
WHERE CURRENT OF <cursor name>
```

告诉 DBMS 更新底层表中与游标中的当前行对应的行的一列或多列中的值。例如下面的 SQL 语句批处理:

```
OPEN cur_payroll_work CURSOR
FETCH FROM cur_payroll_work
FETCH FROM cur_payroll_work INTO @emp_num, @dept, @ot_rate,
@time_in, @time_out, @project, @tcard_hourly_rate,
@tcard_labor_cost, @hrly_payrate, @hours_worked
UPDATE timecards SET tcard_hourly_rate = @hrly_payrate,
tcard_labor_cost = @hrly_payrate * @hours_worked
WHERE CURRENT OF cur_timecards
```

将设置 TIMECARDS 表中生成 CUR_PAYROLL_WORK 游标中第二行(当前行)的 TCARD_HOURLY_RATE 和 TCARD_LABOR_COST 列的值。

技巧 340 使用索引改变游标中行的顺序

在技巧 333“使用 ORDER BY 子句改变游标中行的顺序”中，读者已经学过可以将 ORDER BY 子句添加到游标的 SELECT 语句来对游标中的行按一列或多列的值进行排序。遗憾的是，一些 DBMS 工具包括 MS-SQL Server 都只允许在 READ ONLY 游标的 SELECT 语句中使用 ORDER BY 子句。所以，如果向 MS-SQL Server 上游标的 SELECT 语句添加 ORDER BY 子句，将不能用游标执行定位 DELETE 语句或定位 UPDATE 语句。

如果需要用已排序的数据更新游标，可在游标的基表上创建索引并告诉 DBMS 在填充游标时使用该索引。例如，假设有一组要根据雇员 ID 中的日期与时间排序的考勤卡。如果用 CREATE INDEX 语句的如下句法：

```
CREATE [UNIQUE][CLUSTERED|NONCLUSTERED] INDEX <index name>
ON <table name/view name>
(<column name>[,...<last column name>])
```

在 TIMECARDS 表上创建如下的索引：

```
CREATE INDEX tc_emp_ID_date
ON timecards (emp_ID, time_in, time_out)
```

可以向游标的 SELECT 语句中 FROM 子句添加 WITH INDEX 子句：

```
DECLARE cur_payroll_work CURSOR
FOR SELECT emp_num, dept, ot_rate, time_in, time_out,
project, tcard_hourly_rate, tcard_labor_cost,
COALESCE(hourly_rate, (monthly_salary * 12) / 2080)
hrly_payrate, (CONVERT(REAL(time_out - time_in),8)* 24)
hours_worked
FROM timecards WITH (INDEX(tc_emp_ID_date)), employees
WHERE timecards.emp_ID = employees.emp_ID
FOR UPDATE OF tccard_hourly_rate, tcard.labor_cost
```

来创建行已经被排序（按索引 TC_EMP_ID_DATE 的顺序）的可更新游标。

如果在 DECLARE CURSOR 语句中没有 ORDER BY 子句，DBMS 将把行放到按填充游标时系统添加的顺序排列游标中（响应 OPEN[游标]语句）。SELECT 语句的 SELECT 子句中的 WITH INDEX 子句告诉 DBMS 按 EMP_ID/TIME_IN/TIME_OUT 的顺序为游标提取 TIMECARD 行。

技巧 341 使用 @@FETCH_STATUS 利用 WHILE 循环处理游标中的行

@@FETCH_STATUS 是 MS-SQL Server 定义的包含最近执行的 FETCH 语句执行状态（结果码）的全局变量。@@FETCH_STATUS 值为零（0）表明成功执行了最近的 FETCH 语句。任何其他值则表明 FETCH 语句没有执行成功。同样，可以用 @@FETCH_STATUS 的值作为 Transact-SQL WHILE 循环中的终止测试来操作游标中的行。

例如，如果执行下面的语句批处理：

```
SET NOCOUNT ON
DECLARE cur_authors CURSOR
FOR SELECT * FROM pubs.dbo.authors
--DECLARE temporary variables here
OPEN cur_authors
```

```
FETCH FROM cur_authors
WHILE @@FETCH_STATUS = 0
BEGIN
--Add statements to process individual rows of the cursor
--here
--A FETCH statement without an INTO clause tells the DBMS
--to display the contents of the cursor row's columns to
--the screen.
--If you are processing the rows in a cursor, you will
--normally FETCH cursor row column values INTO temporary
--variables vs. just displaying them.
FETCH FROM cur_authors
END
DEALLOCATE cur_authors
SET NOCOUNT OFF
```

MS-SQL Server 将用 Pubs 数据库中 AUTHORS 表的行填充 CUR_AUTHORS 游标。然后, DBMS 将 FETCH (获取) 并显示游标每行中的值直到 @@FETCH_STATUS 不再是零 (0)。非零的 @@FETCH_STATUS 值表明 DBMS 不能从游标获取另一行, 这会在不再要有要提取的行的假设下终止循环。

请注意当前示例在 WHILE 循环前包含 FETCH 语句。这是因为 @@FETCH_STATUS 的值在连接执行完第一个 FETCH 语句之前没有被定义。而且, 当通过执行 OPEN 语句填充游标时系统没有把 @@FETCH_STATUS 的值设为零 (0)。同样, 必须在 WHILE 循环中测试 @@FETCH_STATUS 的值之前设置它的值。

注意: 对所有被连接使用的游标 @@FETCH_STATUS 都是全局的。同样, 如果在游标上执行 FETCH 语句并随后调用从另一个游标打开并获取行的存储过程, @@FETCH_STATUS 的值将反映在存储过程中执行的最后的 FETCH 语句的执行状态, 而不是在存储过程被调用前执行的 FETCH 语句的执行状态。

技巧 342 使用 DEALLOCATE 语句删除游标并释放其服务器资源

如果完成了使用游标且不准准备通过在当前会话期间重新打开它来重新填充, 可执行 DEALLOCATE 语句来 CLOSE (关闭) 游标同时从数据库删除其结构和定义。与在 DBMS 系统表中留下游标定义与结构的 CLOSE 语句不同, DEALLOCATE 语句不仅删除游标中的任何数据 (通过关闭游标), 而且将游标作为对象从数据库中删除。同样, 如果不准备重新使用游标, 可执行 CLOSE 语句而不是 DEALLOCATE(游标)语句来避免在执行 OPEN 语句在以后重新打开相同的游标前不得不执行另一个 DECLARE CURSOR 语句的麻烦。

DEALLOCATE 语句的句法如下:

```
DEALLOCATE <cursor name>
```

而且游标在被释放前不必关闭——DEALLOCATE 对关闭和释放两个动作都负责处理, 关闭游标 (如果它还打开着) 和删除作为数据库对象的游标。

注意: 释放 SCROLL 型游标会释放游标持有的保护其在底层表中的数据不被其他用户执行的 UPDATE 或 DELETE 语句破坏的滚动锁定。然而, 如果游标是在事务处理中声明与打开

的，释放游标并不会释放任何（仍然）打开的事务处理在游标的底层表中所拥有的锁定。打开的事务处理锁定只有在通过执行 COMMIT 语句（以保存所作的改变）或 ROLLBACK 语句（以撤消任何由事务处理完成的工作）关闭事务处理的时候才被释放。

技巧 343 理解 DECLARE CURSOR 语句的 Transact-SQL 扩展句法

MS-SQL Server 支持两种形式的 DECLARE CURSOR 语句。可以用以下标准的 SQL 句法声明游标：

```
DECLARE <cursor name> [INSENSITIVE][SCROLL] CURSOR  
FOR <SELECT statement>  
[FOR {READ ONLY|UPDATE[OF <column list>]}]
```

（这在技巧 331 “使用 DECLARE CURSOR 语句定义游标”中学习过），或者可以使用以下 Transact-SQL 扩展句法：

```
DECLARE <cursor name> CURSOR [GLOBAL|LOCAL]  
[FORWARD_ONLY|SCROLL][STATIC|KEYSET|DYNAMIC|FAST_FORWARD]  
[READ_ONLY|SCROLL_LOCKS|OPTIMISTIC][TYPE_WARNING]  
FOR <SELECT statement>  
[FOR UPDATE[OF <column list>]]
```

其中：

- GLOBAL，指定游标的作用域对连接是全局的。如果游标有全局作用域，在声明以后就可以在连接上的任何 SQL 语句中被引用（默认情况下，游标通常具有局部作用域，这意味着它作为数据库对象只与声明它的语句批处理、存储过程或触发器一起存在）。DBMS 只有当创建它的连接从服务器断开时才自动 DEALLOCATE（释放）全局型游标。
- LOCAL，指定游标只能被处于定义游标的同样语句批处理、存储过程或触发器中的语句引用。DBMS 在执行语句批处理、触发器或存储过程中最后的语句之后自动释放 LOCAL 型游标——除非存储过程将游标传递给 OUTPUT 参数中的调用者。如果是作为 OUTPUT 参数传递的，系统将在引用游标的最后的变量超出作用域后 DEALLOCATE（释放）游标。
- FORWARD_ONLY，指定用 FETCH NEXT 语句只能经过游标向前移动，一次一行。
- STATIC，是标准 DECLARE CURSOR 语句的 INSENSITIVE 设置的 Transact-SQL 等价物。如果游标是 STATIC（或 INSENSITIVE）的，DBMS 就制作 TEMPDB 数据库表中游标的底层行中数据的临时副本。STATIC 游标不能与 WHERE CURRENT OF 子句一起使用来执行定位 UPDATE（更新）或 DELETE（删除）。而且，STATIC 游标将不会反映在 DBMS 填充游标后对其底层表所作的任何改变。
- KEYSET，指定当 DBMS 打开或填充游标时，游标中被设置的行以及行的顺序。KEYSET 是包含一组惟一识别游标中每一行的一组关键字的 TEMPDB 数据库中的表。KEYSET 游标将反映已经提交的对非关键字列的值所作的任何改动。而且，如果引用的行被从游标的底层表中删除，随后对游标中同一行的 FETCH 操作将返回 @@FETCH_STATUS 的值为 -2。

任何插入到 KEYSET 游标的底层表中的行将不会被添加到游标中——即使新行的列值满

足游标 SELECT 语句中的搜索条件。

对 KEYSET 游标的底层表中关键字列所作的任何改动都被作为 DELETE（删除）原行然后插入新行处理。同样，改变底层表中键列的值的 UPDATE 语句将导致 KEYSET 游标中相应行的“消失”，就像它已经被删除了。通过游标（用定位 UPDATE 语句）对关键字（或非关键字列）所作的任何改动在游标中仍然是可见的。

- DYNAMIC，指定游标要反映由已提交的 UPDATE、DELETE 和 INSERT 对其底层表所作出的任何改变。同样，当满足游标搜索条件的行被添加到游标的底层表时，DBMS 将向游标添加新的行指针。相反，当相应的行被从底层表中删除时或者如果它们的列值被改变而不再满足游标的搜索条件，DBMS 将从游标删除行指针。因为 DYNAMIC 游标中的行可以在执行 FETCH 语句前改变，所以 DYNAMIC 型游标不支持 FETCH ABSOLUTE 语句。
- FAST_FORWARD，指定 DBMS 要把游标优化成 READ_ONLY、FORWARD_ONLY 型游标供以提高性能。
- SCROLL_LOCKS，告诉 DBMS 当服务器向游标添加行时在底层表中的每行上放置一个锁定。SCROLL_LOCKS 选项将用于保证操作游标的 UPDATE 和 DELETE 语句执行成功。
- OPTIMISTIC，指定如果目标行在添加到游标以后被更新，则对游标的任何定位 UPDATE 或 DELETE 操作都将失败。不像 SCROLL_LOCKS 选项，OPTIMISTIC 选项不告诉 DBMS 在每个放到游标中的基础表上都设置锁定。相反，当决定是否允许定位 UPDATE 或 DELETE 操作时，DBMS 会检查底层行的 TIMESTAMP 列的值（或者如果行没有数据类型为 TIMESTAMP 的列，则检查它的校验和）以确定该行在添加到游标中以后是否被改变过。
- TYPE_WARNING，指定如果游标的声明包含有可能把游标从所要求的类型转换成另一种类型的抵触选项，则 DBMS 须向顾客发送警告信息。

技巧 344 使用 @@CURSOR_ROWS 系统变量确定游标中的行数

当 MS-SQL Server 填充除 DYNAMIC 类型以外的任何类型的游标时，DBMS 将 @@CURSOR_ROWS 的值设置为由游标的 SELECT 语句生成的结果表中的行数。例如，如果以下 SELECT 语句：

```
SELECT * FROM pubs.dbo.authors
```

生成含 23 行数据的结果，然后执行语句批处理：

```
DECLARE cur_authors SCROLL CURSOR  
FOR SELECT * FROM pubs.dbo.authors  
PRINT 'There are ' + CAST(@@CURSOR_ROWS AS VARCHAR(6)) +  
' rows in the cursor.'
```

则将显示消息：

```
There are 23 rows in the cursor.
```

请记住 @@CURSOR_ROWS 是全局系统变量，其值是由连接上打开的最后一个游标设置的，其定义如表 16.1 所示。

表 16.1 @@CURSOR_ROWS 的值及其描述

@@CURSOR_ROWS 的值	说明
-<行数>	如果游标是异步填充的，@@CURSOR_ROWS 的值将是代表工作表中当前行数的负数。例如，@@CURSOR_ROWS 的值为-758 说明到目录为止系统仍然正在填充包含 758 行的异步游标。不管是异步还是同步填充，DBMS 对游标的填充一完成，@@CURSOR_ROWS 就将以正值包含游标中的行数
-1	如果游标是 DANAMIC 型的，游标中的行数就可能在 DBMS 打开游标后发生改变。同样，游标的行数被设为-1，因为服务器从来不会确切知道它已经把所有验证过的行提取到游标中
0	还没有游标被打开，游标的 SELECT 语句还没有返回行，或者连接上最后打开的游标已经关闭或释放
<行数>	在 DBMS 完全填充游标以后，@@CURSOR_ROWS 的值将是游标中的行数

第 17 章 理解触发器

技巧 345 理解何时用 CHECK 约束代替触发器

触发器是当用户试图 INSERT（插入）、DELETE（删除）或 UPDATE（更新）表中一行或多行时，DBMS 要执行的一个或多个 SQL 语句的集合。如果试图用 DBMS 强制执行如“不接受任何大于\$100,000 的定单”之类的业务规则，可使用与下面类似的触发器：

```
CREATE trigger order_total_over_100000
ON orders
FOR INSERT, UPDATE
AS IF ((SELECT order_total FROM inserted) > 100000)
BEGIN
PRINT 'Order rejected. Total order > 100000.'
ROLLBACK
END
```

该触发器将显示错误消息并撤消（ROLLBACK）对 ORDERS 表的 ORDER_TOTAL 列的值大于\$100,000 的行的任何 INSERT（插入）或 UPDATE（更新）操作。使用触发器的一个问题是它们作为独立的数据库对象存在，并且如果 ORDERS 表不知道系统将拒绝任何超过\$100,000 的定单，会有人查看定义。所以，人们会发现用 CHECK 约束代替触发器来强制执行简单的业务规则是较为方便直接的。

例如，如要在当前示例中作为 CHECK 约束实现业务规则，可向 CREATE TABLE 语句添加如下的约束：

```
CREATE TABLE orders
(cust_ID INTEGER,
order_date DATETIME,
shipped_date DATETIME,
salesperson_ID INTEGER,
order_total MONEY,
total_paid MONEY,
CONSTRAINT order_total_over_100000
CHECK (order_total <= 100000))
```

技巧 346 理解嵌套游标

一个连接可以有多个同时打开的游标。而且可以用一个游标嵌套另一个游标。然而，当操作外层游标时，将导致多次声明、打开或释放每个内层游标的额外开销。所以，只有需要在内层（被嵌套的）游标的 SELECT 语句中使用外层游标的当前行的一个或多个列值时，才应当在一个游标中嵌套另一个游标的声明。

例如，假设要创建按销售总额从低到高的顺序列出销售办公室并列出每个销售办公室的经理和销售额最高的销售员的存储过程。下列代码用嵌套游标创建生成与图 17.1 所示相似的

输出的存储过程:

```
CREATE PROCEDURE Show_Sales AS
DECLARE @avg_sale MONEY, @emp_fname VARCHAR(20),
@emp_lname VARCHAR(20), @manager INT,
@max_sale MONEY @manager_name VARCHAR(50),
@office_ID INT, @total_sales MONEY
/* cursor used to display the office cursor information,
arranged in ascending order by total sales */
DECLARE cur_office_sales CURSOR
FOR SELECT office, SUM(order_total) total_sales,
AVG(order_total) avg_sale
FROM orders GROUP BY office ORDER BY total_sales DESC
OPEN cur_office_sales
/* work through the office cursor fetching rows until there
are no more rows to display (@@FETCH_STATUS <> 0) */
FETCH cur_office_sales INTO @office_ID, @total_sales,
@avg_sale
WHILE @@FETCH_STATUS = 0
BEGIN
PRINT 'Office ' + CAST(@office_ID AS CHAR(1)) +
' Total sales: ' + CAST(@total_sales AS VARCHAR(11)) +
' Avg sale: ' + CAST(@avg_sale AS VARCHAR(11))
/* the cursor used to list managers for each office is
nested, because it uses the value of @office_ID from the
outer cursor in its SELECT statement */
DECLARE cur_office_manager CURSOR
FOR SELECT emp_ID, office, f_name+ ' l_name office_manager
FROM employees, offices
WHERE emp_ID = manager AND office = @office_ID
OPEN cur_office_manager
/* the cursor used to list the top employee for each office
is also nested, because it too uses the @office_ID value
from the outer cursor */
DECLARE cur_office_employees CURSOR
FOR SELECT f_name, l_name, MAX(order_total) max_order,
AVG(order_total) avg_order
FROM orders, employees
WHERE orders.office = @office_ID AND
salesperson = emp_ID
GROUP BY f_name, l_name ORDER BY max_order DESC
OPEN cur_office_employees
/*work through the (embedded) office manager cursor,
displaying the name and ID of the managers for the
office in the current row in the (outer) office cursor*/
FETCH cur_office_manager INTO @manager, @office_ID,
@manager_name
```

```

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT ' Manager: ' + @manager_name +
    ' |ID: ' + CAST(@manager AS VARCHAR(4)) + '}'
    /* retrieve and display the 'top' employee from the
    (embedded) office employees cursor for the office in the
    current (outer) row of the offices cursor */
    FETCH cur_office_employees INTO @emp_fname, @emp_lname,
    @max_sale, @avg_sale
    PRINT ' Top Salesperson: ' @emp_fname + ' ' +
    @emp_lname + ' Avg sale: ' +
    CAST(@avg_sale AS VARCHAR(11))
    PRINT ' '
    FETCH cur_office_manager INTO @manager, @office_ID,
    @manager_name
END
DEALLOCATE cur_office_manager
DEALLOCATE cur_office_employees
FETCH cur_office_sales INTO @office_ID, @total_sales,
@avg_sales
END
DEALLOCATE cur_office_sales
RETURN 0

```

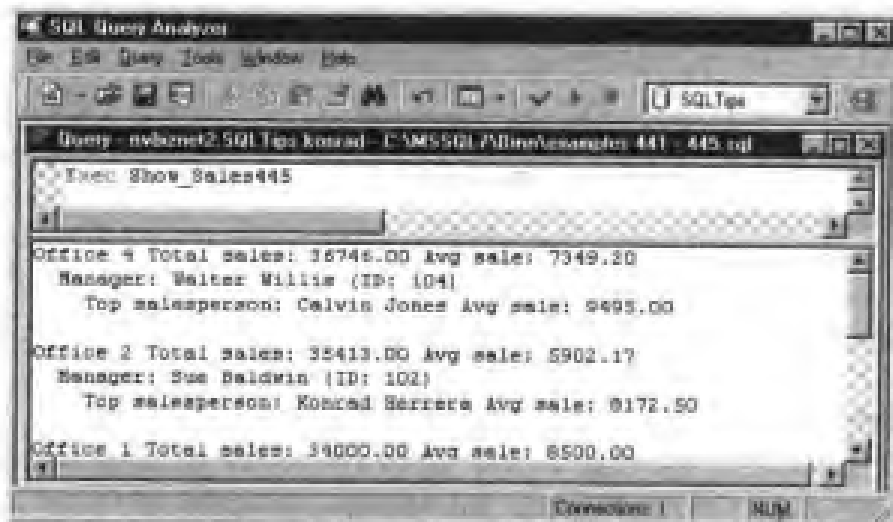


图 17.1 用嵌入式和多次打开的游标生成的组合办事处报告

注意：操作办事处游标时不得不 DEALLOCATE（释放）嵌入式（managers 和 employees）游标的原因是 @OFFICE_ID 的值是在游标被声明时为游标的 SELECT 语句设置的。同样，要改变每个游标的 WHERE 子句中的 @OFFICE_ID 的值以与外层（offices）游标的当前行中 office ID 的值相匹配，必须释放并重新创建嵌入式游标。

技巧 347 理解当前日期和时间的值是在语句开始执行时设置的

GETDATE() 函数返回当前系统日期和时间（如果储存到表的数据类型为 DATETIME 的列

中，则会精确到毫秒)。如果在 SQL 语句中使用 GETDATE() 函数，则 DBMS 只调用该函数一次，而不管系统执行完该语句要花费的时间。

例如，假设要提交如下 UPDATE 语句：

```
SET late_fee = (late_fee +
  ((order_total - amount_paid) * 0.10)),
date_fee_assessed = GETDATE()
WHERE CONVERT(CHAR(9),date_due,6) =
CONVERT(CHAR(9),GETDATE() - 1,6) AND
(order_total - amount_paid) > 0
```

在 01/12/2001 午夜的前一分钟，由于 ORDER 表中巨大的行数，系统直到午夜后 10 分钟才执行完 UPDATE 语句。DBMS 将只按 01/11/2001 的 DUE_DATE 的顺序访问最新的费用——即使当服务器仍然处在执行 UPDATE 语句的过程中时，当前日期从 01/12/2001 变成了 01/13/2001。而且，每个被更新行中的 DATE_FEE_ASSESSED 列都将有相同的日期和时间值（精确到毫秒）——即使每次成功的更新的发生都晚于 DBMS 操作 ORDER 表所用的 11 分钟的那段时间。

技巧 348 用 CREATE TRIGGER 语句创建触发器

只有表的所有者（DBO）或数据库所有者（DBO）才能在表（或其一到多个列）上创建触发器。CREATE TRIGGER 语句的句法如下：

```
CREATE TRIGGER <trigger name>
ON (<table name>|<view name>)
[WITH ENCRYPTION]
{([FOR|AFTER|INSTEAD OF])([INSERT],[UPDATE],[INSERT])}
[NOT FOR REPLICATION]
AS {([IF UPDATE (<column name>)
{([AND|OR] UPDATE (...<last column name>)]
|IF [COLUMNS_UPDATED()
(<bitwise operator>)<column bitmask>)
(...<last comparison operator>
<last column bitmask>)
}]
<SQL statements>
}
```

其中：

- table name | view name 是触发器要结合到的表或视图的名称。
- WITH ENCRYPTION 是在 SYS COMMENTS 表内储存每个触发器的文本的 DBMS。如果 CREATE TRIGGER 语句包含 WITH ENCRYPTION 子句，则 DBMS 将加密触发器的文本，这能阻止用户通过查询 SYS COMMENTS 表显示触发器的代码。然而，加密触发器也会阻止 DBMS 在 MS-SQL Server 复制的过程中把触发器发布到其他 MS-SQL Server。
- FOR 或 AFTER 指定在 DBMS 对列在 ON 子句中的表执行 INSERT、UPDATE 或 DELETE 语句后要执行触发器。

如果 FOR 是指定的惟一关键词, 则 AFTER 是默认的设置。

可以在视图上定义 AFTER 触发器。

- **INSTEAD OF** 告诉 DBMS 当对 CREATE TRIGGER 语句的 ON 子句中指定的表或视图执行的不是 INSERT、UPDATE 或 DELETE 语句时执行触发器。

每个表或视图最多只能有一个 INSTEAD OF (INSERT、UPDATE 和 DELETE) 触发器。

然而, 可以为同一表创建多个视图并且每个视图都可以有不同的 INSTEAD OF (INSERT、UPDATE 和 DELETE) 触发器。

不能在定义内包含 WITH CHECK OPTION 子句的视图上创建 INSTEAD OF 触发器。

- **DELETE、INSERT、UPDATE** 当试图对 ON 子句中列出的表或视图执行操作时, 激活触发器的 SQL 语句。每个 CREATE TRIGGER 语句至少必须包含 3 种类型的语句中的一种以及 3 种语句的任何组合。如果想要 3 种操作中多于一种的操作激活触发器, 就用逗号分开列出想要的触发操作。

不能在定义有 ON DELETE CASCADE 选项的表上创建 INSTEAD OF DELETE 触发器。类似地, 不能在有 ON UPDATE CASCADE 选项的表上创建 INSTEAD OF UPDATE 触发器。

- **NOT FOR REPLICATION** 告诉 DBMS 当表在复制过程中被改变了时, 不要激活触发器。
- **IF UPDATE (<column name>)** 告诉 DBMS 如果只有 INSERT 或 UPDATE 操作修改由 <column name> 指定列名的列中的值时才激活触发器。要在多于一列上测试 INSERT 或 UPDATE 操作, 可用 AND 或 OR 逻辑连接来向 IF UPDATE 子句添加所希望的附加列名。
- **IF COLUMNS_UPDATED()**, COLUMNS_UPDATED() 函数返回说明哪一列已经由于表上的 INSERT 或 UPDATE 操作而被更新的 VARBINARY 位模式。最左边的位是最重要的位, 代表表或视图中的第 1 列, 左边第 2 位代表第 2 列, 左边第 3 位代表第 3 列, 依此类推。例如, 为检查第 2 列、第 4 列或第 6 列是否更新过, 可编写如下的 IF COLUMNS_UPDATED() 子句:

```
IF (COLUMNS)_UPDATED() & 42) > 0
```

或者, 为检查所有 3 列(2、4 和 6)是否更新过, 编写如下的 AS IF COLUMNS_UPDATED() 子句:

```
IF (COLUMNS)_UPDATED() & 42) = 42
```

- **AS <SQL statements>** 详细列出触发器被激活时要执行的操作。无论什么时候 DBMS 激活触发器 (也就是执行触发器的 AS 子句中的 SQL 语句来响应触发器的表或视图上的 INSERT、UPDATE 或 DELETE 操作, DBMS 就创建两个虚拟表 INSERTED 和 DELETED。两个表都与定义触发器的表或视图在结构上是相同的并保存有所有 DBMS 为响应触发器操作而要改变的行的原值与新值。

对 INSERT 触发器而言, INSERTED 保存所有行的要被插入到触发器要结合到的表中的新值。对 UPDATE 触发器而言, INSERTED 则保存要被放到触发器的表中的新 (已经更新过的) 值, 而 DELETED 表则保存 UPDATE 操作前的列值。最终, 对 DELETE 触发器而言, DELETED 表保存要从创建触发器的表中删除的值 (行)。

当不再需要触发器执行一组动作以响应特定表上的 INSERT、DELETE 或 UPDATE 语句时, 可用如下句法执行 DROP TRIGGER 语句删除触发器:

```
DROP TRIGGER <trigger name>[...,<last trigger name>]
```

技巧 349 理解 INSERT 触发器

INSERT 触发器是 DBMS 在特定的表或视图上执行 INSERT 语句后或不是执行 INSERT 语句（AFTER 或 INSTEAD OF）时执行的存储过程。例如，假设销售员无论何时向 ORDERS 表插入一个新定单，都要 UPDATE（更新）OFFICE 表中的 TOTAL_SALES 列和 EMPLOYEES 表中的 TOTAL_SALES 列。则以下 CREATE TRIGGER 语句：

```
CREATE TRIGGER tri_ins_order ON orders
AFTER INSERT AS
SET NOCOUNT ON
UPDATE offices SET total_sales = total_sales +
(SELECT order_total FROM INSERTED)
WHERE offices.office_ID =
(SELECT office_ID FROM INSERTED)
UPDATE employees SET total_sales = total_sales +
(SELECT order_total FROM INSERTED)
WHERE employees.emp_ID =
(SELECT salesperson FROM INSERTED)
```

会告诉 DBMS 在成功执行向 ORDERS 表中插入新定单的 INSERT 语句后，UPDATE（更新）两个表中的总数。

正如在技巧 348“用 CREATE TRIGGER 语句创建触发器”中提到的，当 DBMS 激活 INSERT 触发器时，它会自动创建名为 INSERTED 的虚拟表。如果系统允许完成当前的 INSERT 语句，INSERTED 表就有触发器的 ON 子句的列表中列出的所有列并含有 DBMS 将 INSERT（插入）到触发器表中的每一行的副本。请注意可以在触发器的语句中使用 INSERTED 表中的任何列值。例如，当前示例中的触发器使用要插入的行的 ORDER_TOTAL 列来增加 EMPLOYEES 和 OFFICES 表中 TOTAL_SALES 列的值。而且，触发器使用在每个 UPDATE 语句的 WHERE 子句中插入的行中的 OFFICE_ID 和 SALESPERSON 值指定要两个表中的哪一行。

除了多个表中更新总数以外，还可以用 INSERT 触发器强制执行如“所有定单都必须至少在 EXPECTED_DEL_DATE 之前 3 天进入系统。”之类的业务规则。可用语句：

```
CREATE TRIGGER tri_check_delivery_date ON orders
FOR INSERT AS
SET NOCOUNT ON
IF (SELECT expected_del_date FROM INSERTED) <
(GETDATE() + 3)
BEGIN
ROLLBACK TRAN
RAISERROR('You cannot take an order to be delivered less
than three days from now.',16,1)
END
```

创建的触发器强制执行“在 3 天或更多天以内发送”的业务规则。

正如本技巧的示例中所展示的，可以在相同的表上创建多于一个的触发器。DBMS 把 INSERT 语句和它所激活的所有触发器都看作是相同事务处理的一部分。所以，不管 DBMS 是否在 TRI_CHECK_DELIVERY_DATE 触发器之前执行 TRI_INS_ORDER 触发器，最终的结

果都是相同的。如果新定单上的 EXPECTED_DEL_DATE 早于将来的 3 天，任何由当前打开的事务处理（包括在任何激活的触发器内进行的操作）完成的工作都将被 TRI_CHECK_DELIVERY_DATE 触发器中的 ROLLBACK TRAN 语句撤消（回滚）。

注意：当激活的触发器执行 ROLLBACK 语句时，DBMS 会撤消任何已完成的工作，所以，表上任何附加的触发器都不会被激活，因为将把表返回给它执行任何触发（INSERT、UPDATE 或 DELETE）语句前的条件。

技巧 350 理解 DELETE 触发器

DELETE 触发器是 DBMS 在特定的表或视图上执行 DELETE 语句后或不是执行 DELETE 语句时执行的一种存储过程。例如，如果创建在技巧 349 “理解 INSERT 触发器”中学习过的 INSERT 触发器，在 ORDERS 表上维护 EMPLOYEES 和 OFFICES 表中的 TOTAL_SALES 值，也需要创建 DELETE 触发器，如：

```
CREATE TRIGGER tri_del_order ON orders AFTER DELETE AS
IF @@ROWCOUNT > 1
BEGIN
    ROLLBACK TRAN
    RAISERROR('Each DELETE statement must remove only a single
order.',16,2)
END
ELSE BEGIN
    SET NOCOUNT ON
    UPDATE offices SET total_sales = total_sales -
    (SELECT order_total FROM DELETED)
    WHERE offices.office_ID =
    (SELECT office_ID FROM INSERTED)
    UPDATE employees SET total_sales = total_sales -
    (SELECT order_total FROM DELETED)
    WHERE employees.emp_ID =
    (SELECT salesperson FROM DELETED)
END
```

用户（或应用）程序每次执行从 ORDERS 表删除行的 DELETE 语句时，DBMS 就激活 DELETE 触发器，该触发器按删除的定单的 ORDER_TOTAL 值减少 OFFICES 和 EMPLOYEES 表中 TOTAL_SALES 的值。

请注意与技巧 349 中的触发器不同，DELETE 触发器必须考虑到单个的 DELETE 语句可从触发器表中删除不止一行。当前示例中的 DELETE 触发器用 @@ROWCOUNT 变量确定受触发（DELETE）语句影响的行数。如果 DELETE 语句试图从 ORDERS 表中删除不止一行，DELETE 触发器就通过执行 ROLLBACK 语句撤消删除操作。

虽然当前示例没有允许出现这种情况，还是可以修改该 DELETE 触发器以允许多行删除。请记住，DBMS 会创建虚拟 DELETED 表并用被激活触发器的 DELETE 语句删除的每一行的副本填充。所以，触发器能创建游标，并用 DELETED 表中的行填充，然后以一次一行的方式操作游标——按游标的每一行中 ORDER_TOTAL 列的值从 EMPLOYEES 和 OFFICES 表中减少 TOTAL_SALES 的值。

注意：执行 Transact-SQL TRUNCATE TABLE 语句将从表中删除所有的行而不激活在被截短的表上创建的 DELETE 触发器。这样，当在操作环境中执行 TRUNCATE TABLE 语句时就要小心。在截短含有设计为总计另一个表中数据的 DELETE 触发器的表之后，必须执行一个把由触发器维护的总数归零的 UPDATE 语句。

技巧 351 理解 UPDATE 触发器

UPDATE 触发器是 DBMS 在特定的表或视图上执行 UPDATE 语句后或代替执行 UPDATE 语句时执行的一种存储过程。继续技巧 349“理解 INSERT 触发器”和技巧 350“理解 DELETE 触发器”中的 TOTAL_SALES 简短示例，可以添加如下的触发器：

```
CREATE TRIGGER tri_update_order ON orders AFTER UPDATE AS
DECLARE @rowcount INT
SET @rowcount = @@rowcount
IF UPDATE (order_total) OR UPDATE (office_ID) OR
UPDATE(salesperson) BEGIN
IF @@ROWCOUNT > 1
BEGIN
ROLLBACK TRAN
RAISERROR('Each UPDATE must change only one order at a time.',16,2)
END
ELSE BEGIN
SET NOCOUNT ON
UPDATE offices SET total_sales = total_sales -
(SELECT order_total FROM DELETED)
WHERE offices.office_ID =
(SELECT office_ID FROM DELETED)
UPDATE offices SET total_sales = total_sales +
(SELECT order_total FROM INSERTED)
WHERE offices.office_ID =
(SELECT office_ID FROM INSERTED)
UPDATE employees SET total_sales = total_sales -
(SELECT order_total FROM DELETED)
WHERE employees.emp_ID =
(SELECT salesperson FROM DELETED)
UPDATE employees SET total_sales = total_sales +
(SELECT order_total FROM INSERTED)
WHERE employees.emp_ID =
(SELECT salesperson FROM INSERTED)
END
END
```

以便当用户(或应用程序)改变定单的数量、销售员或所购买货物的部门时，保持 OFFICES 和 EMPLOYEES 表中 TOTAL_SALES 正确的值。

当前示例的 UPDATE 触发器中的第一个 IF 语句确定被更新行的 ORDER_TOTAL、OFFICE_ID 或 SALESPERSON 列的值是否包含在 UPDATE 语句的 SET 子句中。如果 UPDATE 语句没有在这 3 列的任何一列中设定值，对 ORDERS 表中的行的改变就不会影响触发器维护

的 TOTAL_SALES 列的值。所以，触发器仅当这 3 列（ORDER_TOTAL、OFFICE_ID 或 SALESPERSON）中的一或多列中的值是由激活触发器的 UPDATE 语句设定的时，触发器才会执行其 UPDATE 语句。

请注意，UPDATE 触发器可利用 DBMS 激活触发器时创建的 INSERTED 和 DELETED 虚拟表。对 UPDATE 触发器而言，DELETED 表包含每个满足 UPDATE 语句的 WHERE 子句中的搜索条件的行中原列值的副本。与此同时，INSERTED 表则包含相同行的副本，但 INSERTED 表中每一行的列值反映由 UPDATE 语句的 SET 子句所作的改变。

注意：如果 UPDATE 触发器的 IF UPDATE(column)子句出现在 UPDATE 语句的 SET 子句中，则 IF UPDATE 子句返回 TRUE——不管 UPDATE 语句是否改变了列中的值。例如，如果执行下面的 UPDATE 语句：

```
UPDATE orders Set salesperson = salesperson  
WHERE customer_ID = 1001 AND order_date = '01/06/2001'
```

则只要 ORDERS 表中至少有一行的列值满足 UPDATE 语句的 WHERE 子句中的搜索条件，DBMS 就将执行当前示例中的 UPDATE 触发器。（请记住当用户在 OK 按钮上单击鼠标指针时，不管是否发生了改变，有些应用程序都会提交带包含所有可更新的域的 SET 子句的 UPDATE 语句。）如果当列值改变时，触发器完成了大量的工作，可以通过将 DELETED 表中的原列值与 INSERTED 表中更新后的列值对照检查，看用户（或应用程序）是否真的作出了任何改变从而改善性能。触发器仅当列值真的改变了，而且不仅仅是因为出现在 UPDATE 语句的 SET 子句中的触发器列，触发器才需要完成其工作。

与技巧 350 中的 DELETE 触发器一样，UPDATE 触发器必须允许单个 UPDATE 语句有可能设置触发器表中不止一列的值。当前示例中的 UPDATE 触发器使用 @@ROWCOUNT 变量确定受触发(UPDATE)语句影响的行数。如果 UPDATE 语句设定 ORDERS 表中多于一行中的值，UPDATE 触发器会通过执行 ROLLBACK 语句撤消已经完成的工作并返回错误消息和状态码。

虽然当前示例没有允许出现这种情况，还是可以修改该 UPDATE 触发器来允许多行更新。请记住 DELETED 表包含 ORDERS 表每个满足 UPDATE 语句的 WHERE 子句中搜索条件的行中每个原列值的副本。类似地，如果 UPDATE 语句被允许完成其工作，INSERTED 表就像所预期的那样包含带有列值相同行。所以，触发器可能创建两个游标，并用 INSERTED 表中的行填充，其他的行则用 DELETED 表中的行填充，然后操作每个游标，一次一行，按 DELETED 游标中每行的 ORDER_TOTAL 列值减少 TOTAL_SALES 的数值并按 INSERTED 游标中每行的 ORDER_TOTAL 值增加 TOTAL_SALES 的数值。

技巧 352 用 UPDATE 触发器改变 PRIMARY KEY/FOREIGN KEY 对的值

FOREIGN KEY 约束不仅防止删除具有与子表中的一行或多行的 FOREIGN KEY 匹配的 PRIMARY KEY，还防止更新 PRIMARY KEY。

例如，假设正在使用客户的电话号码（在 PHONE_NUMBER 列中）作为 CUSTOMERS 表中的 PRIMARY KEY。DBMS 将通过要求每个新行中的 FOREIGN KEY 值都在 CUSTOMERS 表中一行（而且只有一行）的 PRIMARY KEY 列中有匹配的值来确保没有不存在的客户的定单或支付插入到 ORDERS 和 PAYMENTS 表中。

现在，假设需要改变客户的电话号码。因为新电话号码在 CUSTOMERS 表中不存在，

DBMS 将既不允许改变 ORDERS 表中的电话号码也不允许改变 PAYMENTS 表中的号码。类似地，系统将不允许改变 CUSTOMERS 表中的电话号码，因为这样做将在 ORDERS 和 PAYMENTS 表中创建在 CUSTOMERS 表的 PRIMARY KEY 列中没有相匹配的电话号码的 FOREIGN KEY 值（用旧电话号码）。

幸运的是，可以用 UPDATE 触发器（带 INSTEAD OF 子句）来允许用户执行可改变单个事务处理中的 PRIMARY KEY 和 FOREIGN KEY 两个值的单个 UPDATE 语句。例如，以下 UPDATE 触发器：

```
CREATE TRIGGER tri_change_phone_number ON customers
INSTEAD OF UPDATE AS
IF @@ROWCOUNT > 1
BEGIN
    ROLLBACK
    RAISERROR('You must UPDATE customer columns one row at a time.',16,4)
END
ELSE BEGIN
    SET NOCOUNT ON
    DECLARE @new_phone_num CHAR(7), @old_phone_num CHAR(7)
    SET @new_phone_num = (SELECT phone_number FROM INSERTED)
    SET @old_phone_num = (SELECT phone_number FROM DELETED)
    IF @new_phone_num <> @old_phone_num
    BEGIN
        /*INSERT duplicate customer record with new phone number*/
        INSERT INTO customers SELECT * FROM INSERTED
        /*Change phone number in child rows*/
        UPDATE payments SET phone_number = @new_phone_num
        WHERE phone_number = @old_phone_num
        UPDATE orders SET phone_number = @new_phone_num
        WHERE phone_number = @old_phone_num
        /*DELETE original customer record*/
        DELETE FROM customers
        WHERE phone_number = @old_phone_num
    END
    /*If not changing the primary key then update the remaining
    columns in the customers table as normal*/
    ELSE BEGIN
        UPDATE customers SET
        f_name = (SELECT f_name FROM INSERTED),
        l_name = (SELECT l_name FROM INSERTED)
        WHERE customers.phone_number = @old_phone_num
    END
END
```

首先把一个带有新电话号码的重复的客户行插入到 CUSTOMERS 表中。然后，触发器就把 ORDERS 和 PAYMENTS 表的每个子行中的 FOREIGN KEY 值从旧电话号码改成新电话号码。最后，UPDATE 触发器从 CUSTOMERS 表中删除带有旧电话号码的原先的（现在已经没

有子行) 客户行。

技巧 353 用触发器发送 E-mail 消息

每种 MS-SQL 安装方式都包括一组使 DBMS 能执行外部操作系统外壳中命令字符串以及发送与提取 e-mail 消息的扩展过程。而且, 系统存储过程 SP_ADDEXTENDEDPROC 使得附加扩展存储过程(动态链接库[Dynamic Link Library, DLL]中的函数调用)对服务器也是可用的。触发器可以调用 MS-SQL Server 上可用的任何扩展过程。

例如, 当新项目被插入到 PRODUCTS 表中时为了把电子邮件消息发送到 SALESMANAGER 电子邮箱和 SALESPeople 邮件组的成员, 可创建与下面类似的 INSERT 触发器:

```
CREATE TRIGGER tri_email_re_new_item ON products
FOR INSERT AS
DECLARE @product_code VARCHAR(10),
        @description VARCHAR(30),
        @email_message VARCHAR(75)
SET NOCOUNT ON
/* Get the product code and description of the new product
from the row being inserted into the PRODUCTS table */
SELECT @product_code = INSERTED.product_code,
       @description = INSERTED.description
FROM INSERTED
/* Form the e-mail message to send */
SELECT @email_message = 'PRODUCT CODE: (' + @product_code +
                        ') DESCRIPTION: ' + @description
/* Formulate and send the e-mail message */
EXEC master.dbo.xp_sendmail
@recipients = 'SalesManager, SalesPeople',
@message = @email_message,
@subject = 'NEW PRODUCT Availability Alert'
```

在技巧 461~技巧 463 中将学习更多关于扩展过程 XP_SENDMAIL 的句法(和功能)、如何用其他扩展过程启动与终止 MS-SQL Server 电子邮件处理、如何读取和删除电子邮件消息的知识。现在, 要理解的重要事情是可以使用触发器调用允许发送电子邮件消息的扩展过程。

技巧 354 用 MS-SQL Server Enterprise manager 显示或修改触发器

MS-SQL Server 系统存储过程 SP_HELPTEXT 允许使用句法:

```
SP_HELPTEXT <object name>
```

来显示规则、默认值、未加密的存储过程、触发器、用户定义函数或视图的文本内容。所以, 为了显示控制 TRI_DEL_CUSTOMER 触发器行为的代码, 可调用 SP_HELPTEXT 存储过程如下:

```
SP_HELPTEXT TRI_DEL_CUSTOMER
```

如果要修改(以及查看)触发器的内容, 可通过执行以下步骤用 MS-SQL Server Enterprise manager 显示触发器:

1. 在 Start 按钮上单击。Windows 将显示 Start 菜单。

2. 将鼠标指针移到 Start 菜单中的 Programs 项, 选择 Microsoft SQL Server 2000 选项, 并在 Enterprise Manager 上单击。Windows 将在一个新的应用程序窗口中启动 Enterprise Manager。
3. 在 SQL Server Group 左侧的“+”号上单击显示网络上可用的 MS-SQL Server 的列表。
4. 在带有要显示（以及可能要修改）的触发器的表的 SQL 服务器左侧的“+”号上单击。Enterprise manager 将显示所选择的 SQL Server 的 Databases、Data Transformation、Management、Security 和 Support Services 文件夹。
5. 在 Databases 文件夹左侧的“+”号上单击显示 SQL 服务器上当前数据库的列表, 并随后在带有要显示的触发器的表的数据库左侧的“+”号上单击。对当前项目而言, 是单击 SQLTips 文件夹左侧的“+”号。
6. 为显示在第 5 步中所选择的数据库中表的列表, 在 Tables 图标上单击。Enterprise manager 将用其右窗格显示所选择的数据库中表的列表。
7. 在要查看触发器的表上右击。对当前项目而言是在 CUSTOMER457 左侧的图标上右击。Enterprise manager 将显示上下文敏感的弹出式菜单。
8. 把鼠标指针移到弹出菜单的 All Tasks（所有任务）项上, 然后选择 Manage Triggers（管理触发器）。Enterprise manager 将显示 Trigger Properties（触发器属性）对话框。
9. 在 Name 域右侧的下拉列表按钮上单击, 并从触发器名称的下拉列表中选择要显示的触发器的名称。对当前项目而言, 是在 TRI_DEL_CUSTOMER457 上单击。Enterprise manager 将在 Trigger Properties 对话框的 Text 域中显示触发器的代码, 如图 17.2 所示。



图 17.2 MS-SQL Server Enterprise manager 的 Trigger Properties 对话框

10. 如果要修改触发器的行为, 可在 Trigger Properties 对话框的 Text 窗口中进行所希望的更改、删除和添加, 然后在 Check Syntax 按钮上单击。
 11. 必要时重复步骤 10 以更正 Enterprise manager 对 Text 窗口中的触发器进行句法检查所报告的任何错误。
 12. 要保存所作的更改并使更新过的触发器对 DBMS 可用, 在 OK 按钮上单击。
- 在完成步骤 12 后, Enterprise manager 将把触发器保存为数据库对象, 把触发器的文本

INSERT（插入）到服务器的系统表中，并退出 Trigger Properties 对话框。

注意：如果在用 CREATE TRIGGER 语句创建触发器时包括了 WITH ENCRYPTION 子句，SP_HELPTEXT 或 Enterprise manager 都不能显示触发器的源代码。而且，不能通过在 Trigger Properties 对话框中编辑触发器的文本来使用 Enterprise manager 改变触发器的行为。

技巧 355 用 ALTER VIEW 语句修改视图

与 ALTER TABLE 语句（读者将在技巧 356 “用 ALTER TABLE 语句改变列的数据类型”中学习）不同，不能编写在已有视图中添加、删除一个或多个单独列或改变其类型的 ALTER VIEW 语句。结果，除了用 ALTER 取代单词 CREATE 外，ALTER VIEW 语句的如下句法：

```
ALTER VIEW [<database name>.] [<view owner>.] <view name>
[(<column name>[,...<last column name>])]
[WITH ENCRYPTION|SCHEMABINDING|VIEW_METADATA]
AS <SELECT statement>
[WITH CHECK OPTION]
```

与技巧 153~技巧 158 中所学习的 CREATE VIEW 语句是相同的，所以，如果执行 CREATE VIEW 语句：

```
CREATE VIEW vw_student_name_list
(student_ID, first_name, last_name)
AS SELECT SID, first_name, last_name FROM students
```

并在以后要向视图添加 STUDENTS 表中的 MAJOR 列，应当使用 ALTER VIEW 语句如：

```
ALTER VIEW vw_student_name_list
(student_ID, first_name, last_name, major)
AS SELECT SID, first_name, last_name, major FROM students
```

该语句包括原视图定义中的所有元素，加上要添加到视图的新列。

乍一看，ALTER VIEW 语句似乎没有向 DBMS 数据定义语言添加功能性。毕竟，ALTER VIEW 语句不提供快捷方式来改变视图的定义。就像在本技巧的开头所提到的，用于定义视图的 ALTER VIEW 除了名称（是 ALTER 而不是 CREATE）之外，在所有方面都必须与定义同样视图的 CREATE VIEW 语句相同。所以，在想要改变视图时，还可使用 DROP VIEW 语句删除已有视图，然后用新结构执行 CREATE VIEW 语句重新创建视图看起来是可能的。

然而，ALTER VIEW 语句确实提供一条主要优点（比顺序执行 DROP VIEW/CREATE VIEW 好）在于允许改变视图，而使访问权限及在视图上创建的触发器保持不变。当 DROP（删除）视图时，DBMS 自动删除在视图上创建的任何触发器。然后，当重新创建视图时，系统不仅不恢复视图上的触发器，也不向对原视图拥有权限的用户名（或角色）GRANT（授予）任何访问权限。

同样，如果通过在执行 DROP VIEW 语句之后再执行 CREATE VIEW 语句重新创建视图（以其新的形式）的方法来修改视图，就必须也重新输入原视图上所有触发器的代码，并对视图的访问权限授予允许使用原视图的用户、角色和账号。相反，如果执行 ALTER VIEW 语句对视图定义作出相同的改变，就不必执行 DROP VIEW 语句，原视图上的触发器和所授予的权限在新（修改过的）视图上仍会保留。

技巧 356 用 ALTER TABLE 语句改变列的数据类型

ALTER TABLE 语句的 ALTER COLUMN 子句允许用如下的句法改变列的数据类型：

```
ALTER TABLE <table name>
```

```
ALTER COLUMN <new data type>[(precision[,scale])]
```

其中：

- precision 数中的总位数。
- scale 数中小数点右边的总位数。

例如，假设创建了一个含有数据类型定义为 CHAR(14)的 PHONE_NUMBER 列的 CUSTOMERS 表。只要所有的电话号码都是以数字形式插入到表中的（也就是没有连字符[-]和圆括号[()])，就可以通过执行 ALTER TABLE 语句把 PHONE_NUMBER 列的数据类型转换为 NUMERIC：

```
ALTER TABLE customers ALTER COLUMN phone_number NUMERIC
```

在用 ALTER TABLE 语句改变列的数据类型之前，请检查系统手册看 DBMS 是否在目标列的新或旧数据类型上作了任何限制。

第 18 章 处理 Blobs 数据和文本

技巧 357 理解由二进制和字符大对象 (BLOB) 的处理带来的挑战

因为多数 SQL 服务器被安装成支持商业数据处理, SQL 表中的列最常用于保存这样一些事物, 如名称、地址、销售或存货目录、应收金额、实售金额、实付金额、日期与时间等。保存这些类型数据的列要求的是少量的存储空间——对 INTEGER、MONEY 和 DATETIME 值来说只是几个字节, 而对名称、地址和产品描述 (CHAR 或 VARCHAR) 来说则每个需要 50~100 字节。

由于 (典型的) 最小数据储存需要单独的列, DBMS 产品已经对短行 (根据字节长度) 的操作进行了优化, 即使每个表的行都可能有许多列。SQL 服务器假定当管理磁盘存储空间时行将最多占用几千字节, 并对数据表进行索引以便快速提取。类似地, 编写成操作 SQL 数据库的应用程序假定程序能够将大量的数据行提取到工作站上的可用 (自由) 内存中或通过工作站的内存缓冲区提取到本地硬盘的临时存储区。正如在技巧 298~技巧 329 中使用 Visual Basic 应用程序和 DBLIB 时所看到的一样, 处理查询结果的一次一行的处理技术相对来说易于理解和使用。

在过去的几年中, 面向字符的业务应用程序已经被使用 Windows 图形用户界面的程序所取代。不仅要求应用程序显示高分辨率的图形和动画还要播放声音文件和视频剪辑已经导致“业务”数据项的大小呈指数增长。因此, 由于多媒体应用程序现在已经成为雇员培训计划、销售代理、公司董事会会议上的统计分析、客户支持服务的完整部分, 商业用户需要 DBMS 来管理这些新的、大尺寸的数据类型项目以及其他 (较传统的或尺寸较小的) 字符和数字数据。

对 DBMS 的挑战在于单一的高分辨率图像可能要求成千上万的存储字节, 一个很短的声音文件就将占用几百万字节的磁盘空间, 而视频剪辑通常占用几兆字节的存储空间。所以, 只把一种这样的“新” (多媒体) 数据类型项置于表中一行的一个单独的列可能比一整张有几千行而每行又有许多数字或字符数据的表要求的存储空间还要多。例如, 当视频剪辑不管是存储在单独的磁盘文件中还是表行的一个列中所要求的存储空间是一样的时, 用正常的 DBMS 方法操作过大的行可能会导致一些问题。

例如, 当用户创建并填充带有含视频剪辑列的行的静态游标时, 10 行数据就可能轻易地占用 100MB 的存储空间 (或服务器器内存)。如果只有几个用户, 虽然每个用户 100MB 不成问题, 但在有 100 个或更多活动连接的典型环境中, 用户的查询可能很容易就消耗服务器上几百兆字节的存储空间。而且, 在等待服务器把 (可能的) 几百兆字节的数据从一个磁盘地址复制另一个地址以后, 用户只能用非常大的可用内存来缓冲由 FETCH 语句以一次一行的方式提取的数据, 才能处理工作站上的数据。

最初的 DBMS 试图支持二进制大对象或 BLOB (集团性的, 经常作为 TEXT、NTEXT 和 IMAGE 数据类型项称呼), 涉及在数据库之外以单独的外部文件保存 BLOB 数据。DBMS 在 CHARACTER (或 VARCHAR) 数据类型的列中储存 BLOB 文件的名称而不是 BLOB 数据本身。当应用程序需要 BLOB 数据时, 就从 DBMS 提取 BLOB 文件的名称并随后与操作系统进

行通信以打开并操作文件内容。

遗憾的是，这种方法需要程序员对 DBMS 和文件系统接口都懂。而且，因为应用程序负责管理外部 BLOB 文件的内容，用户不能进行如请求 DBMS 比较两个 BLOB 项（在一个搜索条件内）之类的操作。还有，DBMS 甚至不能提供简单的 BLOB 文本搜索能力。

今天，大多数商业 DBMS 产品都在数据库内储存并管理至少两种类型的 BLOB 数据而不是在外部文件中）。例如，MS-SQL Server 支持 TEXT、NTEXT 和 IMAGE 数据类型列。同时，Oracle 支持同样的 BLOB 数据项，但却将这些数据类型称为 CLOB、NCLOB、BLOB。

技巧 358 理解 MS-SQL Server 的 BLOB（TEXT、NTEXT 和 IMAGE）数据处理过程

MS-SQL Server 允许在一列中存储高达 2GB 的数据作为一个单项。就像其名称所表示的，TEXT 列用于储存字符串数据如储存于数据类型为 CHAR 和 VARCHAR 的列中的数据。所以，当有长度大于 8000 个字符的字符串时，可使用 TEXT 数据类型——8000 是 CHAR 或 VARCHAR 列能够储存的最大字符数。类似地，IMAGE 列允许储存超过 BINARY 或 VARBINARY 列所能储存的 8000 字节的最大长度的二进制数据。储存于数据类型为 IMAGE 的列中的典型项目是 MS-Word 文档、MS-Excel 电子表格、位图（BMP 文件）、Graphics Interchange Format（图形交换格式，GIF）图片和 Joint Photographic Experts Group（联合图像专家组，JPEG 或 JPG）图像。

如果 TEXT 或 IMAGE 列中的数据长度少于或等于 8000 字节，可以用标准的 INSERT、SELECT 和 UPDATE 语句操作这些列。例如，要显示 EMPLOYEES 表中数据类型为 TEXT 的 NOTES 列的内容，如果 NOTES 列有多达 8000 个字符，则可执行如下 SELECT 语句：

```
SELECT notes FROM EMPLOYEES
```

请记住虽然可以用 SELECT 语句显示 IMAGE 列的内容，但联机查询工具（如 MS-Query Analyzer）将以 1 和 0 的形式显示列的二进制码（而不是作为图像、声音文件或视频剪辑显示）。要把二进制的 0 和 1 解释并转换成文档或图像，需要把列的内容发送到应用程序中，如 MS-Word、MS-Excel 或 WinJPEG。

为优化存储空间的使用和数据提取性能，像其他 DBMS 产品一样，MS-SQL Server 按 8k 页的形式组织表中的数据。如果 DBMS 要与行的其他列中的数据混合储存 BLOB 项（TEXT、NTEXT 和 IMAGE 数据），BLOB 就会导致 DBMS 不可能使行数据适应 8k 页。总之，单独的 BLOB 数据项可能需要几百个存储页。为保持表的优化状态，MS-SQL Server 把任何大于 256 字节的 BLOB 数据储存在磁盘上的单独区域。服务器随后把一个指向 BLOB 数据第一页的指针放进表中的 TEXT、NTEXT 或 IMAGE 列。

正如在技巧 357“理解由二进制和字符大对象（BLOB）的处理带来的挑战”中所提到的，大多数操作数据库数据的应用程序不能一次在内存的缓冲区中保存整个 BLOB。相反，程序必须按段处理 BLOB。然而，程序用来提取和储存表数据的 API 以一次一行的方式工作。要解决这个问题，可使用 Active Data Object（ADO）APPENDCHUNK 和 GETCHUNK 方法，这允许在 BLOB 列中以一次一块的方式储存和提取数据（技巧 474“显示保存在 SQL 表内的图像数据”将讨论如何从表中提取图像数据并用 GETCHUNK 方法把它写到磁盘文件中去）。

处理 BLOB 数据要了解的最后一个问题是对事务处理日志要做什么。当记录事务处理中

完成的工作时，DBMS 常常保存有“处理之前”和“处理之后”的映像。因为每个 BLOB 对象都可能非常大，在修改时制作它的“之前”和“之后”的多个副本可能在等待 DBMS 复制数据时导致不可接受的延迟，并可能在这样做时填满事务处理日志。同样，DBMS 产品可既不支持 BLOB 数据日志，也可通过提供启用和禁用日志记录的存储过程（或其他实用程序）来控制日志的记录工作。例如，MS-SQL Server 允许使用 WRITETEXT 和 UPDATETEXT 语句修改 BLOB 的内容而不在事务处理日志中记录“之前”和“之后”的映像。

技巧 359 用 INSERT 或 UPDATE 语句把数据放到 BLOB 数据类型的列中

正如读者在技巧 357 “理解由二进制和字符大对象（BLOB）的处理带来的挑战”和技巧 358 “理解 MS-SQL Server 的 BLOB 数据（TEXT、NTEXT 和 IMAGE）处理过程”中学习过的，MS-SQL Server 允许定义能保存多达 2GB 二进制或字符串数据的表列。这些具有 TEXT、NTEXT 和 IMAGE 数据类型的列通常称为 BLOB（或 MS-SQL Server 上的块数据），可以用 INSERT 语句把数据放到 BLOB 中。

例如，如果有由以下语句创建的 CUSTOMERS 表：

```
CREATE TABLE customers
(cust_ID INTEGER,
f_name VARCHAR(15),
l_name VARCHAR(20),
notes TEXT,
remarks TEXT,
photo IMAGE)
```

可以用下面的 INSERT 语句把数据放到 CUSTOMER 表的 BLOB 列（NOTES、REMARKS、PHOTO）：

```
INSERT INTO customers VALUES (1, 'Konrad', 'King',
'Notes on the customer', 'No Photo as yet', '010')
```

类似地，可以用下面的 UPDATE 语句：

```
UPDATE customers SET remarks = 'The PHOTO column has no
photo. It has only example data used to show the binary
conversion of character string data placed into an IMAGE
column.'
```

```
WHERE cust_ID = 1
```

改变 BLOB 列的内容，就像改变任何其他数据类型列的内容一样。

请记住，DBMS 会把由 INSERT 和 UPDATE 语句完成的所有工作都记录到数据库的事务处理日志中。因为 BLOB 列中的数据量可能非常大（长度可达 2GB），当操作 BLOB 数据时就可能要用 WRITETEXT 语句（参见技巧 360 “用 Transact-SQL WRITETEXT 语句把数据放到 TEXT、NTEXT 或 IMAGE 列中”）或 UPDATETEXT 语句（参见技巧 361 “用 Transact-SQL UPDATETEXT 语句改变 TEXT、NTEXT 或 IMAGE 列的内容”）代替 INSERT 和 UPDATE 语句。由 WRITETEXT 和 UPDATETEXT 语句完成的工作（默认情况下）不会写入事务处理日志。

注意：就像放到当前示例的 REMARKS 列中的“remarks”文本中所提到的，如果在 IMAGE 数据类型的列中储存字符串，服务器将把该字符串看作二进制值。例如，如果执行以下 SELECT 语句：

```
SELECT photo FROM customers
```

来显示把字符串 010 放到当前示例的 IMAGE 数据类型的列 PHOTO 中，MS-SQL Server 将把列的内容显示为：

```
photo
-----
0x303130
```

这是 (0[ASCII 值为 48]1[ASCII 值为 49]0[ASCII 值为 48]) 的十六进制表示。显然，最好在 TEXT 列中储存字符串，并为二进制数据如 MS-Word 文档、MS-Excel 电子表格、图像、声音文件和视频剪辑之类保留二进制列。

技巧 360 用 Transact-SQL WRITETEXT 语句把数据放到 TEXT、NTEXT 或 IMAGE 列中

当用 UPDATE 语句改变 BLOB (TEXT、NTEXT 或 IMAGE 数据类型) 列的内容时，DBMS 必须把列内容的“处理之前”或“处理之后”的副本写入事务处理日志中。为避免过量记录对 BLOB 数据的改变，可执行 Transact-SQL WRITETEXT 语句代替 UPDATE 语句。

WRITETEXT 使用如下句法：

```
WRITETEXT <table.column> <text pointer> <data>
```

其中：

- table.column 是表的名称和要更新的列。
- text pointer 是数据类型为 BINARY(16) 的局部变量，其中包含指向 WRITETEXT 语句的目标 BLOB 数据的第一页的指针。
- data 是 DBMS 要储存在 TEXT、NTEXT 或 IMAGE 列中的数据。

可使用 WRITETEXT 来替换 BLOB (TEXT、NTEXT 或 IMAGE 数据类型) 列中的当前内容。如果要修改（而不是替换）BLOB 的内容，可执行 Transact-SQL UPDATETEXT 语句（读者将在技巧 361 “用 Transact-SQL UPDATETEXT 语句改变 TEXT、NTEXT 或 IMAGE 列的内容”中学习有关知识）。

为了使 DBMS 成功地执行 WRITETEXT 语句，系统需要指向 DBMS 已经分配为保存其 BLOB 数据的第一个 8k 页的有效指针。如果目标列包含非 NULL BLOB，文本指针就已经生效。否则，可通过执行 INSERT 语句或 UPDATE 语句向目标列写入一个 0 长度的字符串或 BLOB 数据的一部分来设置 BLOB 文本指针的值。

例如，为了把一个字符串放到技巧 359 “用 INSERT 或 UPDATE 语句把数据放到 BLOB 数据类型的列中”创建的 CUSTOMERS 表中的 CUST_ID 为 1 的客户行的 REMARKS 列中，可使用与下面类似的语句批处理：

```
UPDATE customers SET remarks = '' WHERE cust_ID = 1
DECLARE @text_pointer BINARY(16)
SELECT @text_pointer = TEXTPTR(remarks)
FROM customers WHERE cust_ID = 1
WRITETEXT customers.remarks @text_pointer
'These are the remarks to write into the REMARKS column'
```

技巧 361 用 Transact-SQL UPDATETEXT 语句改变 TEXT、NTEXT 或 IMAGE 列的内容

正如在技巧 360“用 Transact-SQL WRITETEXT 语句把数据放到 TEXT、NTEXT 或 IMAGE 列中”中所提到的, Transact-SQL WRITETEXT 和 UPDATETEXT 语句都告诉 DBMS 修改 TEXT、NTEXT 或 IMAGE 列 (BLOB) 的内容, 而不向事务处理日志写入“之前”或“之后”的列内容的副本。WRITETEXT 语句只能用于用一个 BLOB 替换另一个 BLOB。与此同时, UPDATETEXT 语句则更灵活, 因为它不仅允许用一个 BLOB 替换另一个 BLOB, 还允许修改 (而不是覆盖) BLOB, 删除某些或所有 BLOB 数据, 以及向 BLOB 添加数据。

UPDATETEXT 语句的句法如下:

```
UPDATETEXT <target table.column> <target text pointer>  
{NULL|<insert offset>} {NULL|<data length>}  
{<data>|(<source table.column> <source text pointer>)}
```

其中:

- target table.column 是内容要被 UPDATETEXT 语句修改的表和列的名称。
- target text pointer 是具有 BINARY(16)数据类型的局部变量, 其中包含指向 UPDATETEXT 语句的目标 BLOB 数据的第一页的指针。
- insert offset 是在插入新数据前要从 TEXT 或 IMAGE 列的已有数据的开始处要跳过的字节数 (对于 TEXT 型数据, <insert offset>是要跳过的 2 字节字符数)。为了在已有数据的起始处插入新数据, 可把<insert offset>设为 0。为了向列的已有内容追加数据, 可把<insert offset>设为 NULL。
- delete length 是要从 BLOB 中<insert offset>开始的地方删除的 TEXT 或 IMAGE 数据的字节数或 NTEXT 数据的 (2 字节) 字符数。如要不删除数据, 就把<delete length>设为 0。要删除 BLOB 的从<insert offset>开始到结尾的所有数据, 可把<delete length>设为 NULL。
- data 是 DBMS 要插入到<target table.column>的<insert offset>处的数据类型为 CHAR、NCHAR、VARCHAR、NVARCHAR、BINARY、VARBINARY、TEXT、NTEXT 或 IMAGE 的实际值或变量。
- source table.column 是要被插入到<target table.column>的<insert offset>处的表或列的名称。
- source text pointer 是 BINARY(16)数据类型的局部变量, 其中包含指向要插入到<target table.column>的<insert offset>处的<source table.column>中的 BLOB 的第一页的指针。

为了使 DBMS 成功地执行 UPDATETEXT 语句, 系统需要指向要更新的 BLOB 的第一页的有效指针。如果目标列不是 NULL, 指向其 BLOB 的文本指针就已经生效。相反, 如果 BLOB 数据列是 NULL, 就必须通过执行 INSERT 语句或 UPDATE 语句把一个长度为 0 的字符串或 BLOB 数据的一部分写入目标列来设置 BLOB 文本指针的值。

例如, 为了在 customer 1 的 REMARKS 列中当前 BLOB 的开头插入字符串, 可使用与下面类似的语句批处理处理:

```
DECLARE @dest_text_pointer BINARY(16)
```

```

SELECT @dest_text_pointer = TEXTPTR(remarks)
FROM customers WHERE cust_ID = 1
IF @dest_text_pointer IS NULL
BEGIN
UPDATE customers SET remarks = '' WHERE cust_ID = 1
SELECT @dest_text_pointer = TEXTPTR(remarks)
FROM customers WHERE cust_ID = 1
END
IF @dest_text_pointer IS NOT NULL
UPDATETEXT customers.remarks @dest_text_pointer 0 0
' This is an additional remark.'

```

技巧 362 用 READTEXT()函数读取 TEXT、NTEXT 或 IMAGE 列中的部分（或全部）数据

Transact-SQL READTEXT()函数是面向 BLOB（TEXT、NTEXT 或 IMAGE 数据）的，就像 SUBSTRING()是面向字符串的一样。SUBSTRING()函数返回字符串中的一部分字符，而 READTEXT()函数从 BLOB 返回指定的字节数。

如果要显示 BLOB 中的第一个@@TEXTSIZE 字节，可以使用在 SELECT 子句中列出 BLOB 列的 SELECT 语句（读者将在技巧 366 “理解 TEXTSIZE 选项和@@TEXTSIZE()函数”中学习如何设置 TEXTSIZE 选项中的字节数）。然而，如果要读取数目不同于@@TEXTSIZE 的一些字节或从不同于 BLOB 第一字节的地方开始，就要使用 READTEXT 而不是 SELECT 语句。

READTEXT 的句法如下：

```

READTEXT <table.column> <text pointer> <offset> <size>
[HOLDLOCK]

```

其中：

- **table.column** 是有要显示的 TEXT、NTEXT 或 IMAGE 数据的表和列。
- **text pointer** 是 BINARY(16)数据类型的局部变量，其中包含指向 DBMS 要显示的 <table.column>中的 BLOB 的第一页的指针。
- **offset** 是开始读操作之前要跳过的 TEXT 或 IMAGE 数据的字节数。（如果正在处理 NTEXT BLOB 数据，则 offset 是要跳过的 2 字节字符的数目。
- **size** 在开始读取操作前要跳过的 TEXT 或 IMAGE 数据的字节数。
- **HOLDLOCK** 告诉 DBMS 读取时 LOCK（锁定）BLOB 直到处理完毕。如果在 BLOB 上放置 HOLDLOCK，其他用户仍然能够“读取”BLOB 中的数据，但是他们不能对其进行修改，直到 COMMIT（提交）或 ROLLBACK（回滚）执行 READTEXT 语句的事务处理。

所以，要显示从 CUST_ID 为 1 的 CUSTOMERS 表行的 REMARKS 列中从第 15 字节开始的 21 字节数据，可执行以下语句批处理：

```

DECLARE @text_pointer BINARY(16)
SELECT @text_pointer = TEXTPTR(remarks)
FROM customers WHERE cust_ID = 1
IF @text_pointer IS NOT NULL

```

```
READTEXT customers.remarks @text_pointer 14 21
```

技巧 363 用 MS-SQL Server 的 TEXTVALID()函数确定文本指针是否有效

正如读者在技巧 359 “用 INSERT 或 UPDATE 语句把数据放到 BLOB 数据类型的列中”和技巧 362 “用 READTEXT()函数读取 TEXT、NTEXT 或 IMAGE 列中的部分（或全部）数据”中看到的，WRITETEXT、UPDATETEXT 和 READTEXT 语句要求提供目标 TEXT、NTEXT 或 IMAGE 数据的第一个 8k 页的地址。TEXTVALID()允许检查 TEXTPTR()函数是否将返回指向目标表的特殊列中的 BLOB 数据的有效文本指针。

TEXTVALID()函数使用的句法如下：

```
TEXTVALID(<'table.column'>,<text pointer>)
```

其中：

- table.column 是 TEXTVALID()函数要检查其文本指针值的 TEXT、NTEXT 或 IMAGE 数据类型的表和列。
- text pointer 是指向储存在<table.column>中的 TEXT、NTEXT 或 IMAGE 数据的第一个 8k 页的指针。

如果<table.column>的<text pointer>有效，TEXTVALID()将返回 INTEGER 值 1，或者如果<text pointer>无效，则返回 INTEGER 值 0。同样，UPDATE 语句：

```
UPDATE customers SET remarks = ''
```

```
WHERE TEXTVALID('customers.remarks', TEXTPTR(remarks)) <> 1
```

使用 TEXTVALID()函数检查指向 CUSTOMERS 表中每一行的 REMARKS BLOB 的文本指针的有效性，并用 REMARKS 列中的 NULL 值初始化这些行的文本指针。

注意：当在 TEXT、NTEXT 或 IMAGE (BLOB) 数据类型列中插入带 NULL 值的行时，DBMS 并不预先分配一个 8k 的页来保存 BLOB 的第一个 8k 数据。结果，如果 BLOB 列值是 NULL，指向第一个 8k 存储页的指针就是无效的（或 NULL）。把值（即使是 0 长度的字符串）一放进 BLOB 列，DBMS 就分配一个 8k 的页并初始化指向它的文本指针。BLOB 数据项的文本指针会保持有效，直到用户或应用程序删除储存 BLOB 的行。

技巧 364 用 PATINDEX()函数返回 BLOB 中第一次出现的地址

PATINDEX()函数允许搜索 TEXT 和 NTEXT 列的内容从而找出一个字符串。如果 PATINDEX()在列的数据中找到字符串，就返回字符串中第一个字符的第一次出现的字节的偏移量（也就是位置）。相反，如果函数在列中没有找到字符，就返回 0。仅当 TEXT 或 NTEXT 列是 NULL 或告诉函数搜索 NULL 值，PATINDEX()才返回 NULL。

PATINDEX()函数的句法如下：

```
PATINDEX(<'<pattern>'>,<expression>)
```

其中：

- pattern 是 DBMS 要在 CHAR、VARCHAR、TEXT 或 NTEXT <expression>中查找的字符串实际值。<pattern>可以包括与任何数目的字符匹配的百分号（%）通配符，或者与任何单个字符匹配的下划线（_）通配符。
- expression 是要在其中搜索由<pattern>给定的字符串第一次出现的位置的 CHAR、VARCHAR、TEXT 或 NTEXT 数据类型的表达式（通常为列名）。

例如要得到投诉“鸡肉”产品的客户的列表,可在查询的 WHERE 子句中使用 PATINDEX() 函数,如:

```
SELECT cust_ID, f_name + ' ' + l_name 'Customer Name',
       complaints
FROM customers WHERE PATINDEX('%chicken%', complaints) >= 1
```

<pattern> (要在 BLOB 中查找的字符串)两端的百分号(%)告诉 DBMS 看作匹配<pattern> 前或后的任何数目的字符。如果只要显示那些以<pattern>开头的行,则省略前导百分号(%), 如下所示:

```
SELECT cust_ID, f_name + ' ' + l_name 'Customer Name',
       complaints
FROM customers WHERE PATINDEX('chicken%', complaints) >= 1
```

为了只显示那些带有以<pattern>结尾的 COMPLAINTS 列的行,则省略结尾的百分号(%), 如下所示:

```
SELECT cust_ID, f_name + ' ' + l_name 'Customer Name',
       complaints
FROM customers WHERE PATINDEX('%chicken', complaints) >= 1
```

技巧 365 用 DATALENGTH() 函数返回 BLOB 中的字节数

DATALENGTH() 函数的如下句法:

DATALENGTH(<expression>)

将以整数形式返回任何数据类型的<expression>中的字节数。如果表达式有 NULL 值, DATALENGTH() 将返回 NULL。

DATALENGTH() 允许确定 TEXT、NTEXT 或 IMAGE 列中 BLOB 的大小。例如,为了显示 Pubs 数据库的 PUB_INFO 表中 LOGO 和 PR_INFO 列(数据类型分别为 IMAGE 和 TEXT) 的字节数,可执行如下的查询:

```
USE pubs
SET TEXTSIZE 45
SELECT DATALENGTH(logo) 'Logo Size',
       DATALENGTH(pr_info) 'PR Info Size',
       pr_info 'Public Relations Info Text' FROM pub_info
```

而且 MS-SQL Server 将显示与图 18.1 所示的类似结果。

Logo Size	PR Info Size	Public Relations Info Text
440	41071	This is sample text data for Mac Millan Books.
440	41071	This is sample text data for Prentice & Hall.
440	41071	This is sample text data for Algonquin Software.
440	41071	This is sample text data for Five Lakes Press.
440	41071	This is sample text data for Addison Wesley.
440	41071	This is sample text data for O'Reilly, publisher.
440	41071	This is sample text data for No Starch Books.
440	41071	This is sample text data for No Starch Books.

图 18.1 显示用 DATALENGTH() 函数显示 IMAGE 和 TEXT 列的字节长度的查询结果的 MS-SQL Server Query Analyzer 窗口

注意: TEXTSIZE 选项 (将在技巧 366 “理解 TEXTSIZE 选项和 @@TEXTSIZE() 函数” 中学习) 设置当被通知显示 TEXT、NTEXT 或 IMAGE 列的内容时 SELECT 语句将返回的最大字符数。

技巧 366 理解 TEXTSIZE 选项和 @@TEXTSIZE() 函数

当使用交互程序如 MS-SQL Server 的 Query Analyzer 时, 最好防止 TEXT 数据从一行折返到另一行以查询结果表能适合屏幕上的单一行。或者, 假设编写一个操作有 TEXT 数据类型的表中的数据的应用程序, 并且只对列中的前 250 个字符感兴趣。在这两种情况下, 可以执行 SET 语句指定 SELECT 语句 (交互式执行或通过应用程序) 将从数据类型为 TEXT、NTEXT 或 IMAGE (例如, 包含 BLOB 的列) 的列中返回的字节数。

SET 语句的 TEXTSIZE 选项允许告诉 MS-SQL Server 当被告知显示 BLOB 时要返回的的字节数。用户每次登录时, 服务器都把连接的初始 TEXTSIZE 值设置成 64,512——意思是 BLOB 上的 SELECT 语句将最多返回 64,512 字节的数据。而要告诉服务器只返回前 20 字节数据, 可执行 SET 语句:

```
SET TEXTSIZE 20
```

既可使用 PRINT 语句, 也可使用 SELECT 语句显示会话的 TEXTSIZE 选项的值。例如, PRINT 语句:

```
PRINT @@TEXTSIZE
```

将显示数字 64512——如果还没有用 SET 语句改变连接的 TEXTSIZE 选项的初始值的话。

注意: 当设定 TEXTSIZE 的值以指定想要 SELECT 语句从 TEXT 或 NTEXT 数据类型的列中显示的字符数, 请记住每个 NTEXT (或 UNICODE) 字符占用 2 字节的存储空间。同样, 例如, 如果把 TEXTSIZE 设为 20, SELECT 语句将显示一列 TEXT 类型的数据中的前 20 个字符, 并且只有前 10 个字符属于 NTEXT 类型的数据列。

第 19 章 使用 MS-SQL Server 信息架构视图

技巧 367 理解信息架构

数据库由数据和元数据组成。顾名思义，数据是储存在组成数据库的表中的每一行的列中值（包括 TEXT、NTEXT 和 IMAGE 项）的集合。元数据是对数据库对象的描述并用来储存和管理数据库中的数据项。系统在一组通称为系统目录的表中储存数据库的元数据。如果查询系统目录中的表，就可以显示描述数据库的表、列、视图、约束、域、权限等的元数据。简而言之，DBMS 系统目录的表包含描述数据库中每个对象的元数据。

虽然 DBMS 为了它自己内部的用途而维护系统目录，但也可以用标准的 SQL 查询（SELECT 语句）访问系统表。事实上，当在技巧 304 “使用 SqlColName() 函数提取由查询生成的结果集中的列名”中调用 SQLColName() 函数提取查询结果集中列的名称时，以及在技巧 303 “使用 SqlNumCols() 函数确定由查询生成的结果集中的列数”中调用 SqlNumCols() 函数提取查询返回的列数时，就使用了系统目录。简而言之，用户可访问的系统目录使得关系 DBMS 成为自我描述的并允许设计通用目的“前端”如 MS-SQL Server 的 Query Analyzer 那样的查询工具。

虽然所有主要的 DBMS 都使用系统目录储存元数据，SQL-89 和当前的 SQL-92 标准并不要求这样做。结果，包含在其中的系统目录表和元数据的结构在不同的 DBMS 产品之间变化很大。避开承担试图使 DBMS 开发商改变其产品并就系统目录的标准结构（和物理实现）达成一致的不可能的任务，SQL 标准委员会定义了一系列系统目录表的视图。这些视图被称为 SQL-92 标准中的信息架构。

构成信息架构的系统目录表视图给予每个用户一个标准化的方法获得用户的数据库连接上可用的数据库对象的描述。技巧 368~技巧 387 描述了 20 个 MS-SQL Server 上可用的信息架构视图。现在要理解的重要事情是，如果一个 DBMS 产品遵循 SQL-92 标准，它也将有与在 MS-SQL Server 上所找到的信息架构视图类似的信息架构视图。然而，SQL-92 标准允许 DBMS 开发商向标准定义的信息架构视图添加额外的视图和额外的列。所以，用下面的 20 个技巧（技巧 368~技巧 387）作为对各种信息架构视图中可用的信息类型的指南，并检验为该 DBMS 产品特别定义的信息架构的系统手册。

技巧 368 理解信息架构的 CHECK_CONSTRAINTS 视图

CHECK_CONSTRAINTS 视图是一个基于来自 SYSOBJECTS 和 SYSCOMMENTS 系统表信息的虚拟表，其中为当前数据库中每个 CHECK 约束都包括一行有关信息量。当执行以下查询时：

```
SELECT * FROM INFORMATION_SCHEMA.CHECK_CONSTRAINTS
```

DBMS 将返回如表 19.1 所示的关于当前数据库中的对象上的 CHECK 约束的信息，对于当前数据库来说，用来与服务器连接的登录 ID 至少有一种访问权限。

表 19.1 信息架构的 CHECK_CONSTRAINTS 视图中的列

列名	数据类型	描述
CONSTRAINT_CATALOG	NVARCHAR(128)	定义约束的数据库
CONSTRAINT_SCHEMA	NVARCHAR(128)	约束的所有者
CONSTRAINT_NAME	SYSNAME	约束名称
CHECK CLAUSE	NVARCHAR(4000)	CHECK 约束的文本

为了显示当前数据库（甚至是没有任何访问权限的数据库）中所有表的信息架构的 CHECK_CONSTRAINTS 视图的列信息，可执行如下 SELECT 语句：

```
SELECT db_name() 'Database/Catalog',
       user_name(sysobjects.UID) 'Constraint Owner',
       sysobjects.name 'Constraint Name',
       syscomments.text 'Constraint Text'
FROM sysobjects, syscomments
WHERE sysobjects.ID = syscomments.ID
AND sysobjects.xtype = 'C'
```

技巧 369 理解信息架构的 COLUMN_DOMAIN_USAGE 视图

COLUMN_DOMAIN_USAGE 视图是一个基于来自 SYSOBJECTS、SYSCOLUMNS 和 SYSTYPES 系统表信息的虚拟表，其中为当前数据库中具有用户定义的数据类型的每一列都包括一行有关信息。当执行如下查询时：

```
SELECT * FROM INFORMATION_SCHEMA.COLUMN_DOMAIN_USAGE
```

DBMS 返回如表 19.2 所示的关于当前数据库的表中用户定义的数据类型的列的信息，而用来与服务器连接的登录 ID 对此表至少有一种访问权限。

表 19.2 信息架构的 COLUMN_DOMAIN_USAGE 视图中的列

列名	数据类型	描述
DOMAIN_CATALOG	NVARCHAR(128)	创建用户定义的数据类型的数据库
DOMAIN_SCHEMA	NVARCHAR(128)	创建用户定义的数据类型的用户名
DOMAIN_NAME	SYSNAME	用户定义的数据类型的名称
TABLE_CATALOG	NVARCHAR(128)	包含带有用户定义的数据类型的列的表的数据库
TABLE_SCHEMA	NVARCHAR(128)	带有用户定义的数据类型的列的表的所有者的用户 ID
TABLE_NAME	SYSNAME	带有用户定义的数据类型的列的表的名称
COLUMN_NAME	SYSNAME	带用户定义的数据类型的列的名称

为了显示当前数据库（甚至是没有任何访问权限的数据库）中所有表的信息架构的 COLUMN_DOMAIN_USAGE 视图的列信息，可执行如下 SELECT 语句：

```
SELECT db_name() 'Domain Database/Catalog',
       user_name(systypes.UID) 'Domain Owner',
       systypes.name 'Domain Name',
```

```

db_name() 'Table Database/Catalog'
user_name(sysobjects.UID) 'Table Owner',
sysobjects.name 'Table Name',
syscolumns.name 'Column Name'
FROM sysobjects, syscolumns, systypes
WHERE sysobjects.ID = syscolumns.ID
AND syscolumns.xusertype = systypes.xusertype
AND systypes.xusertype > 256

```

技巧 370 理解信息架构的 COLUMN_PRIVILEGES 视图

COLUMN_PRIVILEGES 视图是一个基于来自 SYSPROTECTS、SYSOBJECTS 和 SYSCOLUMNS 系统表信息的虚拟表, 其中对一列所授予的或由数据库用户授予的每种权限都包括一行有关信息。当执行如下查询时:

```
SELECT * FROM INFORMATION_SCHEMA.COLUMN_PRIVILEGES
```

DBMS 返回如表 19.3 所示的关于由与服务器连接的登录 ID 授予或授予该登录 ID 的对表列操作的权限的信息。

表 19.3 信息架构的 COLUMN_PRIVILEGES 视图中的列

列名	数据类型	描述
GRANTOR	NVARCHAR(128)	授予权限的账号 ID
GRANTEE	NVARCHAR(128)	被授予权限的账号 ID
TABLE_CATALOG	NVARCHAR(128)	包含授予对表列操作的权限的数据库的名称
TABLE_SCHEMA	NVARCHAR(128)	拥有某一表的的用户 ID, 该表中包括
TABLE_NAME	SYSNAME	表列的表的名称
COLUMN_NAME	SYSNAME	对其授予操作权限的列的名称
PRIVILEGE_TYPE	VARCHAR(10)	在列上被授予的权限 (许可)
IS_GRANTABLE	VARCHAR(3)	如果被授予权限者有授予权限 (许可) 给其他人的权力, 则值为 YES; 否则, 如果授权者没有此权力, 则值为 NO

为了查看定义信息架构的 COLUMN_PRIVILEGES 视图的 Transact-SQL 代码, 可执行在技巧 388 “用 MS-SQL Server Enterprise Manager 查看信息架构视图的内容” 中概述的过程。

技巧 371 理解信息架构的 COLUMNS 视图

COLUMNS 视图是一个基于来自 SYSOBJECTS、SYSTYPES、SYSCOLUMNS、MASTER.DBO.SPT_DATATYPE_INFO、SYSCOMMENTS 和 MASTER.DBO.SYSCHARSETS 系统表信息的虚拟表, 其中对每种授予数据库用户或数据库用户授予别人的对的列操作的权限都包括一行有关信息。当执行如下查询时:

```
SELECT * FROM INFORMATION_SCHEMA.COLUMNS
```

DBMS 返回如表 19.4 所示的关于当前数据库中的表的列的信息, 此信息对与服务器连接的登录 ID 是可用的。

表 19.4 信息架构的 COLUMNS 视图中的列

列名/数据类型	描述
TABLE_CATALOG NVARCHAR(128)	包含该列的数据库名称
TABLE_SCHEMA NVARCHAR(128)	拥有包含该列的表的用户的 ID
TABLE_NAME NVARCHAR(128)	包含该列的表的名称
COLUMN_NAME NVARCHAR(128)	列名
ORDINAL_POSITION SMALLINT	表定义中列的位置, 从左到右计算 (左边第一列的序数位置为 1)
COLUMN_DEFAULT NVARCHAR(4000)	列的默认值
IS_NULLABLE VARCHAR(3)	如果列允许有 NULL 值, 则为 YES; 否则为 NO
DATA_TYPE NVARCHAR(128)	列的数据类型
CHARACTER_MAXIMUM_LENGTH SMALLINT	BINARY、CHARACTER、TEXT 和 IMAGE 数据的最大字符长度; 具有其他数据类型的列则为 NULL
CHARACTER_OCTET_LENGTH SMALLINT	BINARY、CHARACTER、TEXT 和 IMAGE 数据的最大字节长度; 具有其他数据类型的列则为 NULL
NUMERIC_PRECISION TINYINT	INTEGER 数据、精确的和近似的数字数据、和 MONEY 数据的精度; 具有其他数据类型的列则为 NULL
NUMERIC_PRECISION_RADIX SMALLINT	INTEGER 数据、精确的和近似的数字数据、和 MONEY 数据的精度基数; 具有其他数据类型的列则为 NULL
NUMERIC_SCALE TINYINT	INTEGER 数据、精确的和近似的数字数据、和 MONEY 数据的大小; 具有其他数据类型的列则为 NULL
DATETIME_PRECISION SMALLINT	DATETIME 和 INTERVAL 数据的子类型代码; 具有其他数据类型的列则为 NULL
CHARACTER_SET_CATALOG VARCHAR(6)	字符和 TEXT 数据的字符集所处的数据库 (通常为 MASTER) 的名称; 具有其他数据类型的列则为 NULL
CHARACTER_SET_SCHEMA VARCHAR(3)	字符和 TEXT 数据的字符集的所有者的 ID (通常为 DBO); 具有其他数据类型的列则为 NULL
CHARACTER_SET_NAME NVARCHAR(128)	字符和 TEXT 数据的字符集的名称; 具有其他数据类型的列则为 NULL
COLLATION_CATALOG VARCHAR(6)	定义列的字符和 TEXT 数据的排序次序所处的数据库 (通常为 MASTER) 的名称; 具有其他数据类型的列则为 NULL
COLLATION_SCHEMA VARCHAR(3)	含字符和 TEXT 数据的列的排序规则的所有者的 ID (通常为 DBO); 具有其他数据类型的列则为 NULL
COLLATION_NAME NVARCHAR(128)	含有字符和 TEXT 数据的列的排序次序的名称; 具有其他数据类型的列则为 NULL
DOMAIN_CATALOG NVARCHAR(128)	如果列有用户定义的数据类型时创建用户定义的数据类型所在的数据库的名称; 具有其他数据类型的列则为 NULL
DOMAIN_SCHEMA NVARCHAR(128)	如果列有用户定义的数据类型时创建用户定义的数据类型的用户的 ID; 具有其他数据类型的列则为 NULL
DOMAIN_NAME NVARCHAR(128)	如果列有用户定义的数据类型时用户定义的数据类型的名称; 具有其他数据类型的列则为 NULL

为了查看定义信息架构的 COLUMNS 视图的 Transact-SQL 代码，可执行在技巧 388 “用 MS-SQL Server Enterprise Manager 查看信息架构视图的内容”中概述的过程。

技巧 372 理解信息架构的 CONSTRAINT_COLUMN_USAGE 视图

CONSTRAINT_COLUMN_USAGE 视图是一个基于来自 SYSOBJECTS、SYSCOLUMNS 和 SYSTYPES 系统表信息的虚拟表，其中的带有定义其上的约束的每一列都包括一行有关信息。当执行如下查询时：

```
SELECT * FROM INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE
```

DBMS 将返回如表 19.5 所示的关于当前数据库中定义了约束的表、列和约束的信息，而且与服务器连接的登录 ID 对此数据库对象至少有一种访问权限。

表 19.5 信息架构的 CONSTRAINT_COLUMN_USAGE 视图中的列

列名/数据类型	描述
TABLE_CATALOG NVARCHAR(128)	有约束列在表中定义的数据库名
TABLE_SCHEMA NVARCHAR(128)	内含受约束的列的表的所有者的 ID
TABLE_NAME NVARCHAR(128)	列被约束的表的名称
COLUMN_NAME NVARCHAR(128)	被约束的列的名称
CONSTRAINT_CATALOG NVARCHAR(128)	定义了约束的数据库名
CONSTRAINT_SCHEMA NVARCHAR(128)	约束所有者的 ID
CONSTRAINT_NAME NVARCHAR(128)	约束名

为显示有约束的当前数据库（甚至是没有任何访问权限的数据库）中所有列的 CONSTRAINT_COLUMN_USAGE 视图的信息的信息架构，可执行如下 SELECT 语句：

```
SELECT iskcu.table_catalog 'Table Database',
       iskcu.table_schema 'Table Owner ID',
       iskcu.table_name 'Table Name',
       iskcu.column_name 'Column Name',
       iskcu.constraint_catalog 'Constraint Database',
       iskcu.constraint_schema 'Constraint Owner ID',
       iskcu.constraint_name 'Constraint Name'
FROM information_schema.key_column_usage iskcu
UNION
SELECT db_name(),
       user_name(tableobj.UID),
       tableobj.name,
       syscolumns.name,
       db_name(),
       user_name(constraintobj.UID),
       constraintobj.name
FROM sysobjects tableobj, sysobjects constraintobj,
     syscolumns
WHERE tableobj.ID = constraintobj.parent_obj
AND constraintobj.xtype = 'C'
AND constraintobj.info = syscolumns.colid
AND syscolumns.ID = constraintobj.parent_obj
```

```

UNION
SELECT db_name(),
user_name(tableobj.UID),
tableobj.name,
syscolumns.name,
db_name(),
user_name(constraintobj.UID),
constraintobj.name
FROM sysobjects tableobj, sysobjects constraintobj,
syscolumns, systypes
WHERE tableobj.ID = syscolumns.ID
AND syscolumns.xusertype = systypes.xusertype
AND systypes.xusertype > 255
AND systypes.domain = constraintobj.ID
AND constraintobj.xtype = 'R'

```

技巧 373 理解信息架构的 CONSTRAINT_TABLE_USAGE 视图

CONSTRAINT_TABLE_USAGE 视图是一个基于 SYSOBJECTS 系统表信息的虚拟表，其中为每个定义了约束的表都包括一行有关信息。当执行如下查询时：

```
SELECT * FROM INFORMATION_SCHEMA.CONSTRAINT_TABLE_USAGE
```

DBMS 将返回如表 19.6 所示的关于当前数据库中定义了约束的每个表的信息，而且用来与服务器连接的登录 ID 对此表至少有一种访问权限。

表 19.6 信息架构的 CONSTRAINT_TABLE_USAGE 视图中的列

列名	数据类型	描述
TABLE_CATALOG	NVARCHAR(128)	定义被约束的表所处的数据库名
TABLE_SCHEMA	NVARCHAR(128)	定义被约束的表的所有者的 ID
TABLE_NAME	SYSNAME	定义被约束的表的名称
CONSTRAINT_CATALOG	NVARCHAR(128)	定义表的约束所处的数据库名
CONSTRAINT_SCHEMA	NVARCHAR(128)	约束所有者的 ID
CONSTRAINT_NAME	SYSNAME	表的约束的名称

为了显示有约束的当前数据库（甚至是没有任何访问权限的数据库）中所有表的信息架构的 CONSTRAINT_TABLE_USAGE 视图的信息，可执行如下 SELECT 语句：

```

SELECT db_name() 'Table Database',
user_name(tableobj.UID) 'Table Owner',
tableobj.name 'Table Name',
db_name() 'Constraint Database',
user_name(constraintobj.UID) 'Constraint Owner',
constraintobj.name 'Constraint Name'
FROM sysobjects tableobj, sysobjects constraintobj
WHERE tableobj.ID = constraintobj.parent_obj
AND constraintobj.xtype IN ('C','DQ','PK','F')

```

技巧 374 理解信息架构的 DOMAIN_CONSTRAINTS 视图

DOMAIN_CONSTRAINTS 视图是一个基于来自 SYSTYPES 系统表信息的虚拟表，其中为有 RULE 绑定的当前数据库中的每种用户定义的数据类型都包括一行有关信息。当执行如下查询时：

```
SELECT * FROM INFORMATION_SCHEMA.DOMAIN_CONSTRAINTS
```

DBMS 将返回如表 19.7 所示的关于与服务器连接的登录 ID 有访问权限的当前数据库中每个绑定了 RULE（规则）的用户定义的数据类型的信息。

表 19.7 信息架构的 DOMAIN_CONSTRAINTS 视图中的列

列名	数据类型	描述
CONSTRAINT_CATALOG	NVARCHAR(128)	定义 RULE（规则）所处的数据库名
CONSTRAINT_SCHEMA	NVARCHAR(128)	RULE（规则）的所有者的 ID
CONSTRAINT_NAME	SYSNAME	RULE（规则）的名称
DOMAIN_CATALOG	SYSNAME	定义用户定义的数据类型所处的数据库
DOMAIN_SCHEMA	NVARCHAR(128)	创建用户定义的数据类型的用户的 ID
DOMAIN_NAME	SYSNAME	用户定义的数据类型的名称
IS_DEFERRABLE	VARCHAR(2)	不管约束检查是否可以推迟，总是为 NO
INITIALLY_DEFERRED	VARCHAR(2)	不管最初约束检查是否可以推迟，总是为 NO

为了显示所有绑定了 RULE（规则）的用户定义的数据类型（甚至是那些无访问权限者）的信息架构的 DOMAIN_CONSTRAINTS 视图的信息，可执行如下 SELECT 语句：

```
SELECT db_name() 'Rule Database/Catalog',
user_name(sysojects.UID) 'Rule Owner',
sysojects.name 'Rule Name',
db_name() 'User Data Type Database',
user_name(systypes.UID) 'User Data Type Owner',
systypes.name 'User Data Type Name',
'NO' 'Is Deferrable',
'NO' 'Initially Deferred'
FROM sysojects, systypes
WHERE sysojects.xtype = 'R'
AND sysojects.ID = systypes.Domain
AND systypes.xusertype > 256
```

技巧 375 理解信息架构的 DOMAINS 视图

DOMAINS 视图是一个基于来自 MASTER.DBO.SPT_DATATYPE_INFO、SYSTYPES、SYSCOMMENTS 和 MASTER.DBO.SYSCHARSETS 系统表信息的虚拟表，其中为当前数据库中每种用户定义的数据类型都包括一行有关信息。当执行如下查询时：

```
SELECT * FROM INFORMATION_SCHEMA.DOMAINS
```

DBMS 将返回如表 19.8 所示的关于与服务器连接的登录 ID 有访问权限的当前数据库中的每个用户定义的数据类型的信息。

表 19.8 信息架构的 DOMAINS 视图中的列

列名/数据类型	描述
DOMAIN_CATALOG NVARCHAR(128)	定义用户定义的数据类型所处的数据库名
DOMAIN_SCHEMA NVARCHAR(128)	创建用户定义的数据类型的用户的 ID
DOMAIN_NAME SYSNAME	用户定义的数据类型的名称
DATA_TYPE SYSNAME	系统 (SQL) 提供的数据类型
CHARACTER_MAXIMUM_LENGTH SMALLINT	有字符、BINARY、TEXT 和 IMAGE 数据的列的最大字符长度；其他数据类型的列则为 NULL
CHARACTER_OCTET_LENGTH SMALLINT	有字符、BINARY、TEXT 和 IMAGE 数据的列的最大字节长度；其他数据类型的列则为 NULL
COLLATION_CATALOG VARCHAR(6)	有字符或 TEXT 数据的列的排序次序所处的数据库名（通常为 MASTER）；其他数据类型的列则为 NULL
COLLATION_SCHEMA VARCHAR(3)	有字符或 TEXT 数据的列的数据库名（通常是排序次序的所有者的 ID）——总是为 DBO，或者对其他数据类型的列则为 NULL
COLLATION_NAME NVARCHAR(128)	有字符或 TEXT 数据的列的排序次序的名称；其他数据类型的列则为 NULL
CHARACTER_SET_CATALOG VARCHAR(6)	为含字符或 TEXT 数据的列定义排序次序所处的数据库名——总是为 MASTER，或者对其他数据类型的列则为 NULL
CHARACTER_SET_SCHEMA VARCHAR(3)	含字符或 TEXT 数据的列的字符集的所有者的 ID——总是为 DBO，或者对其他数据类型的列则为 NULL
CHARACTER_SET_NAME NVARCHAR(128)	带数字数据的列的字符集的名称；其他数据类型的列则为 NULL
NUMERIC_PRECISION TINYINT	带数字数据的列的精度；其他数据类型的列则为 NULL
NUMERIC_PRECISION_RADIX SMALLINT	带数字数据的列的精度基数；其他数据类型的列则为 NULL
NUMERIC_SCALE TINYINT	含数字数据的列的大小；其他数据类型的列则为 NULL
DATETIME_PRECISION SMALLINT	DATETIME 和 INTERVAL 数据的分型代码；其他数据类型的列则为 NULL
DOMAIN_DEFAULT NVARCHAR(4000)	域的 Transact-SQL 定义

为了查看定义信息架构的 DOMAINS 视图的 Transact-SQL 代码，可执行在技巧 388 “用 MS-SQL Server Enterprise Manager 查看信息架构视图的内容”中概述的过程。

技巧 376 理解信息架构的 KEY_COLUMN_USAGE 视图

KEY_COLUMN_USAGE 视图是一个基于来自 SYSOBJECTS、SYSCOLUMNS、SYSREFERENCES、MASTER.DBO.SPT_VALUES 和 SYSINDEXES 系统表信息的虚拟表，其中为每个用在 PRIMARY KEY 或 FOREIGN KEY 中的列都包括一行信息。当执行如下查询时：

```
SELECT * FROM INFORMATION_SCHEMA.DOMAINS
```

DBMS 将返回如表 19.9 所示的关于与服务器连接的登录 ID 至少有一种访问权限当前数据库中的作为表中 PRIMARY KEY 或 FOREIGN KEY 的一部分的列的信息。

表 19.9 信息架构的 KEY_COLUMN_USAGE 视图中的列

列名/数据类型	描述
CONSTRAINT_CATALOG NVARCHAR(128)	包含用于列的 PRIMARY KEY 或 FOREIGN KEY 约束的数据库名
CONSTRAINT_SCHEMA NVARCHAR(128)	拥有用于列的键约束的用户的 ID
CONSTRAINT_NAME NVARCHAR(128)	拥有用于列的键约束的名称
TABLE_CATALOG NVARCHAR(128)	包含具有用于列的键约束的表的数据库名
TABLE_SCHEMA NVARCHAR(128)	拥有具有用于列的键约束的表的用户的 ID
TABLE_NAME NVARCHAR(128)	具有用于列的键约束的表的名称
COLUMN_NAME NVARCHAR(128)	处于 PRIMARY KEY 或 FOREIGN 键约束之中的列的名称
ORDINAL_POSITION INTEGER	列在定义它的表中的位置

为了查看定义信息架构的 KEY_COLUMN_USAGE 视图的 Transact-SQL 代码，可执行在技巧 388 “用 MS-SQL Server Enterprise Manager 查看信息架构视图的内容”中概述的过程。

技巧 377 理解信息架构的 PARAMETERS 视图

PARAMETERS 视图是一个基于来自 SYSOBJECTS、SYSCOLUMNS 和 MASTER.DBO.SYSCCHARSETS 系统表信息的虚拟表，其中为每个从传递给用户定义的函数或存储过程的参数都包括一行信息，而且视图为每个用户定义包括具有函数的返回值信息的附加行。当执行如下查询时：

```
SELECT * FROM INFORMATION_SCHEMA.PARAMETERS
```

DBMS 将返回如表 19.10 所示的关于用户定义的函数和过程的参数和每个对与服务器连接的登录 ID 可用函数的返回值的信息。

表 19.10 信息架构的 PARAMETERS 视图中的列

列名/数据类型	描述
SPECIFIC_CATALOG VARCHAR(128)	包含使用 PARAMETERS 视图的当前行中描述的参数或返回在其中描述的值的函数或过程（程序）的数据库名
SPECIFIC_SCHEMA NVARCHAR(128)	使用参数或返回值的程序的所有者的 ID
SPECIFIC_NAME NVARCHAR(128)	使用参数或返回值的程序的名称
ORDINAL_POSITION SMALLINT	参数在程序的参数表（从 1 开始）中的位置；或者对函数的返回值来说则为 0
PARAMETER_MODE NVARCHAR(10)	输入参数为 IN，输出参数为 OUT；输入/输出参数为 INOUT
IS_RESULT NVARCHAR(10)	如果参数是函数的返回值则为 YES；否则为 NO
AS_LOCATOR NVARCHAR(10)	如果参数声明为定位符则是 YES；否则为 NO
PARAMETER_NAME NVARCHAR(128)	参数名；如果参数是函数的返回值，则为 NULL

续表

列名/数据类型	描述
DATA_TYPE NVARCHAR(128)	参数的数据类型
CHARACTER_MAXIMUM_LENGTH INTEGER	字符和 TEXT 数据的最大字符数；其他数据类型的列则为 NULL
CHARACTER_OCTET_LENGTH INTEGER	包含字符和 TEXT 数据的列的最大字节数；其他数据类型的列则为 NULL
COLLATION_CATALOG NVARCHAR(128)	包含一种字符数据类型的参数的排序次序的数据库名；其他数据类型的列则为 NULL
COLLATION_SCHEMA NVARCHAR(128)	包含一种字符数据类型的参数的排序规则（排序次序）的计划名；其他数据类型的列则为 NULL
COLLATION_NAME NVARCHAR(128)	含有一种字符数据类型的参数的排序规则（排序次序）的名称；其他数据类型的列则为 NULL
CHARACTER_SET_CATALOG NVARCHAR(128)	包含有一种字符数据类型的参数的字符集的数据库名；其他数据类型的列则为 NULL
CHARACTER_SET_SCHEMA NVARCHAR(128)	含有一种字符数据类型的参数的字符集的所有者的用户 ID；其他数据类型的列则为 NULL
CHARACTER_SET_NAME NVARCHAR(128)	含有一种字符数据类型的参数的字符集的名称；其他数据类型的列则为 NULL
NUMERIC_PRECISION TINYINT	具有数字数据的列的精度；其他数据类型的列则为 NULL
NUMERIC_PRECISION_RADIX SMALLINT	具有数字数据的列的精度基数；其他数据类型的列则为 NULL
NUMERIC_SCALE TINYINT	具数字数据的的列的大小；其他数据类型的列则为 NULL
DATETIME_PRECISION SMALLINT	DATETIME 和 INTERVAL 数据的分型代码；其他数据类型的列则为 NULL
INTERVAL_TYPE NVARCHAR(30)	NULL（保留给将来使用）
INTERVAL_PRECISION SMALLINT	NULL（保留给将来使用）
USER_DEFINED_TYPE_CATALOG NVARCHAR(128)	NULL（保留给将来使用）
USER_DEFINED_TYPE_SCHEMA NVARCHAR(128)	NULL（保留给将来使用）
USER_DEFINED_TYPE_NAME NVARCHAR(128)	NULL（保留给将来使用）
SCOPE_CATALOG NVARCHAR(128)	NULL（保留给将来使用）
SCOPE_SCHEMA NVARCHAR(128)	NULL（保留给将来使用）
SCOPE_NAME NVARCHAR(128)	NULL（保留给将来使用）

为了查看定义信息架构 PARAMETERS 视图的 Transact-SQL 代码，可执行在技巧 388“用 MS-SQL Server Enterprise Manager 查看信息架构视图的内容”中概述的过程。

技巧 378 理解信息架构的 REFERENTIAL_CONSTRAINTS 视图

REFERENTIAL_CONSTRAINTS 视图是一个基于来自 SYSOBJECTS、SYSREFERENCES

和 SYSINDEXES 系统表信息的虚拟表，其中为当前数据表中的每个 FOREIGN KEY 约束都包括一行信息。当执行如下查询时：

```
SELECT * FROM INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS
```

DBMS 将返回如表 19.11 所示的关于表中每个 FOREIGN KEY 约束的信息。

表 19.11 信息架构的 REFERENTIAL_CONSTRAINTS 视图中的列

列名/数据类型	描述
CONSTRAINT_CATALOG NVARCHAR(128)	定义 FOREIGN KEY 所在的数据库名
CONSTRAINT_SCHEMA NVARCHAR(128)	FOREIGN KEY 约束的所有者的用户 ID
CONSTRAINT_NAME SYSNAME	FOREIGN KEY 约束的名称
UNIQUE_CONSTRAINT_CATALOG NVARCHAR(128)	定义 FOREIGN KEY 所在的数据库名
UNIQUE_CONSTRAINT_SCHEMA NVARCHAR(128)	FOREIGN KEY 约束的所有者的用户 ID
UNIQUE_CONSTRAINT_NAME SYSNAME	FOREIGN KEY 的名称
MATCH_OPTION VARCHAR(7)	总是为 NONE
UPDATE_RULE VARCHAR(9)	根据如果 PRIMARY KEY 的值被改变时，DBMS 是否要改变 FOREIGN KEY 的值以与它所引用的 PRIMARY KEY 的新值相匹配，其值可以是 CASCADE 或 NO ACTION
DELETE_RULE VARCHAR(9)	根据如果具有被 FOREIGN 键引用的 PRIMARY KEY 的行被删除时，DBMS 是否要删除具有 FOREIGN KEY 的行，其值可以是 CASCADE 或 NO ACTION

为显示当前数据库中外键（甚至是那些位于没有访问权限的表中的）的信息架构的 REFERENTIAL_CONSTRAINTS 视图的信息，可执行如下 SELECT 语句：

```
SELECT db_name() 'FOREIGN KEY Database',
user_name(foreignkey_obj.UID) 'FOREIGN KEY Owner',
foreign_obj.name 'FOREIGN KEY Table',
db_name() 'PRIMARY KEY Database',
user_name(primarykey_obj.UID) 'PRIMARY KEY Owner',
sysindexes.name 'FK INDEX Name',
'NONE' 'Match',
CASE WHEN (OBJECTPROPERTY
(sysreferences.constid, 'CnstIsUpdateCascade')=1)
THEN 'CASCADE'
ELSE 'NO ACTION' END 'UPDATE Rule',
CASE WHEN (OBJECTPROPERTY
(sysreferences.constid, 'CnstIsDeleteCascade')=1)
THEN 'CASCADE'
ELSE 'NO ACTION' END 'DELETE Rule'
FROM sysobjects foreignkey_obj, sysreferences, sysindexes,
sysobjects primarykey_obj
```

```

WHERE foreignkey_obj xtype = 'F'
AND sysreferences.constID = foreignkey_obj.ID
AND sysreferences.rkeyID = sysindexes.ID
AND sysreferences.rkeyindID = sysindexes.indID
AND sysreferences.rkeyID = primarykey_obj.ID

```

技巧 379 理解信息架构的 ROUTINES 视图

ROUTINES 视图是一个基于来自 SYSOBJECTS、SYSCOLUMNS、MASTER.DBO.SPT_DATATYPE_INFO 和 MASTER.DBO.SYSCHARSETS 系统表信息的虚拟表，其中为当前数据库中的每个存储过程和函数都包括一行信息。当执行如下查询时：

```
SELECT * FROM INFORMATION_SCHEMA.ROUTINES
```

DBMS 将返回如表 19.12 所示的关于能通过与服务器连接的登录 ID 访问的当前数据库中的存储过程和函数的信息。

表 19.12 信息架构的 ROUTINES 视图中的列

列名/数据类型	描述
SPECIFIC_CATALOG NVARCHAR(128)	定义存储过程或函数所在的数据库
SPECIFIC_SCHEMA	定义存储过程或函数的所有者的用户 ID
SPECIFIC_NAME NVARCHAR(128)	定义存储过程或函数的名称
ROUTINE_CATALOG NVARCHAR(128)	同 SPECIFIC_CATALOG
ROUTINE_SCHEMA NVARCHAR(128)	同 SPECIFIC_SCHEMA
ROUTINE_NAME NVARCHAR(128)	同 SPECIFIC_NAME
ROUTINE_TYPE NVARCHAR(20)	对存储过程其值为 PROCEDURE，对函数其值为 FUNCTION
MODULE_CATALOG NULL NVARCHAR(128)	NULL（保留给将来用）
MODULE_SCHEMA NULL NVARCHAR(128)	NULL（保留给将来用）
UDT_CATALOG NULL NVARCHAR(128)	NULL（保留给将来用）
UDT_SCHEMA NULL NVARCHAR(128)	NULL（保留给将来用）
UDT_NAME NULL NVARCHAR(128)	NULL（保留给将来用）
DATA_TYPE NVARCHAR(128)	由函数返回的值的数据类型；如函数返回值为表，则值为 TABLE；对存储过程，值为 NULL
CHARACTER_MAXIMUM_LENGTH INTEGER	返回字符类型数据的函数的最大字符长度；否则为 NULL
CHARACTER_OCTET_LENGTH INTEGER	返回字符类型数据的函数的最大字节长度；否则为 NULL
COLLATION_CATALOG NVARCHAR(128)	包含返回字符类型数据的函数的排序规则（排序次序）的数据库名；否则为 NULL
COLLATION_SCHEMA NVARCHAR(128)	返回字符类型数据的函数的排序规则（排序次序）的所有者的用户 ID；否则为 NULL

续表

列名/数据类型	描述
COLLATION_NAME NVARCHAR(128)	返回字符类型数据的函数的排序规则（排序次序）的名称；否则为 NULL
CHARACTER_SET_CATALOG NVARCHAR(128)	包含返回字符类型数据的函数的字符集的数据库名；否则为 NULL
CHARACTER_SET_SCHEMA NVARCHAR(128)	包含返回字符类型数据的函数的字符集的数据库的所有者的 ID（通常为 DBO）；否则为 NULL
CHARACTER_SET_NAME NVARCHAR(128)	返回字符类型数据的函数的字符集的名称；否则为 NULL
NUMERIC_PRECISION SMALLINT	由返回数字数据的函数返回的值的精度；否则为 NULL
NUMERIC_PRECISION_RADIX SMALLINT	由返回数字数据的函数返回的值的精度基数；否则为 NULL
NUMERIC_SCALE SMALLINT	由返回数字数据的函数返回的值的的大小；否则为 NULL
DATETIME_PRECISION SMALLINT	返回 DATETIME 值的函数的 DATETIME 值中的秒的小数位数；否则为 NULL
INTERVAL_TYPE NULL NVARCHAR(30)	NULL（保留给将来用）
INTERVAL_PRECISION NULL SMALLINT	NULL（保留给将来用）
TYPE_UDT_CATALOG NULL NVARCHAR(128)	NULL（保留给将来用）
TYPE_UDT_SCHEMA NULL NVARCHAR(128)	NULL（保留给将来用）
TYPE_UDT_NAME NULL NVARCHAR(128)	NULL（保留给将来用）
SCOPE_CATALOG NULL NVARCHAR(128)	NULL（保留给将来用）
SCOPE_SCHEMA NULL NVARCHAR(128)	NULL（保留给将来用）
SCOPE_NAME NULL NVARCHAR(128)	NULL（保留给将来用）
MAXIMUM_CARDINALITY NULL BIGINT	NULL（保留给将来用）
DTD_IDENTIFIER NULL NVARCHAR(128)	NULL（保留给将来用）
ROUTINE_BODY NVARCHAR(30)	总是为 SQL
ROUTINE_DEFINITION NVARCHAR(4000)	定义存储过程或函数的 Transact-SQL 语句——如果没有加密的话；否则为 NULL
EXTERNAL_NAME NULL NVARCHAR(128)	NULL（保留给将来用）
EXTERNAL_LANGUAGE NULL NVARCHAR(30)	NULL（保留给将来用）
PARAMETER_STYLE NULL NVARCHAR(30)	NULL（保留给将来用）
IS_DETERMINISTIC NVARCHAR(10)	对确定性的函数为 YES；而对非确定性函数和存储过程则为 NULL
SQL_DATA_ACCESS NVARCHAR(30)	对所有函数为 READS，而对所有存储过程则为 MODIFIES

续表

列名/数据类型	描述
IS_NULL_CALL NVARCHAR(10)	总是为 YES
SQL_PATH NULL NVARCHAR(128)	NULL (保留给将来用)
SCHEMA_LEVEL_ROUTINE NVARCHAR(10)	总是为 YES
MAX_DYNAMIC_RESULT_SETS SMALLINT	对函数为 0, 而对存储过程则为-1
IS_USER_DEFINED_CAST NVARCHAR(10)	总是为 NO
IS_IMPLICITLY_INVOCABLE NVARCHAR(10)	总是为 NO
CREATED DATETIME	创建函数或存储过程的日期和时间
LAST_ALTERED INTEGER	最近修改函数或存储过程的日期和时间

为了查看定义信息架构的 ROUTINES 视图的 Transact-SQL 代码, 可执行在技巧 388 “用 MS-SQL Server Enterprise Manager 查看信息架构视图的内容”中概述的过程。

技巧 380 理解信息架构的 SCHEMATA 视图

SCHEMATA 视图是一个基于来自 MASTER.DBO.SYSDATABASES 和 MASTER.DBO.SYSCHEMSETS 系统表信息的虚拟表, 其中为当前 MS-SQL Server 上每个可访问的数据库都包括一行信息。例如, 如果执行如下 SELECT 语句:

```
SELECT * FROM INFORMATION_SCHEMA.ROUTINES
```

则 DBMS 将返回如表 19.13 所示的关于由 DBMS 管理的每个数据库的信息。

表 19.13 信息架构 SCHEMATA 视图中的列

列名	数据类型	描述
CATALOG_NAME	NVARCHAR(128)	数据库名
SCHEMA_NAME	NVARCHAR(128)	数据库拥有者的用户 ID——通常为 DBO
SCHEMA_OWNER	NVARCHAR(128)	同 SCHEMA_NAME
DEFAULT_CHARACTER_ SET_CATALOG	VARCHAR(6)	总是为 NULL
DEFAULT_CHARACTER_ SET_SCHEMA	VARCHAR(3)	总是为 NULL
DEFAULT_CHARACTER_ SET_NAME	SYSNAME	默认字符集的名称

技巧 381 理解信息架构的 TABLE_CONSTRAINTS 视图

TABLE_CONSTRAINTS 视图是一个基于来自 SYSOBJECTS 系统表信息的虚拟表, 其中为当前数据库中每个表约束都包括一行信息。当执行如下查询时:

```
SELECT * FROM INFORMATION_SCHEMA.ROUTINES
```

DBMS 将返回如表 19.14 所示的关于当前数据库中表约束的信息。

表 19.14 信息架构的 TABLE_CONSTRAINTS 视图中的列

列名/数据类型	描述
CONSTRAINT_CATALOG NVARCHAR(128)	包含 CHECK、FOREIGN KEY、PRIMARY KEY 或 UNIQUE 约束的数据库名（例如，包含表约束的数据库名）
CONSTRAINT_SCHEMA NVARCHAR(128)	拥有表约束的用户的 ID
CONSTRAINT_NAME SYSNAME	表约束的名称
TABLE_CATALOG NVARCHAR(128)	包含在其上定义了表约束的表的数据库名
TABLE_SCHEMA NVARCHAR(128)	拥有带表约束的表的用户的 ID
TABLE_NAME SYSNAME	其内容受表约束所约束的表的名称
CONSTRAINT_TYPE VARCHAR(11)	表约束的类型：CHECK、FOREIGN KEY、PRIMARY KEY 或 UNIQUE 约束
IS_DEFERRABLE VARCHAR(2)	总是为 NO
INITIALLY_DEFERRED VARCHAR(2)	总是为 NO

为了显示当前数据库中所有表约束（甚至那些没有访问权限的表上的约束）的信息架构的 TABLE_CONSTRAINTS 视图的信息，可执行如下 SELECT 语句：

```
SELECT db_name() 'Constraint Database',
       user_name(constraintobj.UID) 'Constraint Owner',
       constraintobj.name 'Constraint Name',
       db_name() 'Table Database',
       user_name(tableobj.UID) 'Table Owner',
       tableobj.name 'Table Name',
       CASE constraintobj.xtype
       WHEN 'C' THEN 'CHECK'
       WHEN 'UQ' THEN 'UNIQUE'
       WHEN 'PK' THEN 'PRIMARY KEY'
       WHEN 'F' THEN 'FOREIGN KEY'
       END 'Constraint Type',
       'NO' 'Is Deferrable',
       'NO' 'Initially Deferred'
FROM sysobjects constraintobj, sysobjects tableobj
WHERE tableobj.ID = constraintobj.parent_obj
AND constraintobj.xtype IN ('C', 'UQ', 'PK', 'F')
```

技巧 382 理解信息架构的 TABLE_PRIVILEGES 视图

TABLE_PRIVILEGES 视图是一个基于来自 SYSPROTECTS 和 SYSOBJECTS 系统表的信息基础的虚拟表，其中为授予用户或由用户授予的当前数据库中每个表的每种操作权限都包括一行信息。当执行如下查询时：

```
SELECT * FROM INFORMATION_SCHEMA.TABLE_PRIVILEGES
```


DBMS 返回如表 19.15 所示的关于授予用来与服务器连接的登录 ID 或由用来与服务器连接的登录 ID 授予的每个表上的权限的信息。

表 19.15 信息架构的 TABLE_PRIVILEGES 视图中的列

列名	数据类型	描述
GRANTOR	NVARCHAR(128)	授予权限的账号的用户 ID
GRANTEE	NVARCHAR(128)	被授予权限的账号的用户 ID
TABLE_CATALOG	NVARCHAR(128)	在其上被授予的权限的表的名称
TABLE_SCHEMA	NVARCHAR(128)	表所有者的用户 ID
TABLE_NAME	SYSNAME	表的名称
PRIVILEGE_TYPE	VARCHAR(10)	被授予的权限类型
IS_GRANTABLE	VARCHAR(3)	如果被授权者可以授权给另一用户, 则为 YES; 否则为 NO

为了查看定义信息架构 TABLE_PRIVILEGES 视图的 Transact-SQL 代码, 可执行在技巧 388 “用 MS-SQL Server Enterprise Manager 查看信息架构视图的内容” 中概述的过程。

技巧 383 理解信息架构的 TABLES 视图

TABLES 视图是一个基于来自 SYSOBJECTS 系统表信息的虚拟表, 其中为当前数据库中的每个表都包括一行信息。当执行如下查询时:

```
SELECT * FROM INFORMATION_SCHEMA.TABLES
```

DBMS 将返回如表 19.16 所示的关于当前数据库中的每个表的信息。

表 19.16 信息架构的 TABLES 视图中的列

列名	数据类型	描述
TABLE_CATALOG	NVARCHAR(128)	定义表所在的数据库名
TABLE_SCHEMA	NVARCHAR(128)	拥有表的用户的 ID
TABLE_NAME	SYSNAME	表的名称
TABLE_TYPE	VARCHAR(10)	表的类型; 可以是 VIEW 或 BASE TABLE

为了显示当前数据库中所有表 (包括那些在没有访问权限的表中者) 的信息架构的 TABLES 视图的信息, 可执行如下 SELECT 语句:

```
SELECT db_name() 'Database/Catalog Name',
user_name(sysobjects.UID) 'Table Owner',
sysobjects.name 'Table Name',
CASE sysobjects.xtype
WHEN 'U' THEN 'BASE TABLE'
WHEN 'V' THEN 'VIEW'
END 'Table Type'
FROM sysobjects
WHERE sysobjects.xtype IN ('U','V')
```

技巧 384 理解信息架构的 VIEW_COLUMN_USAGE 视图

VIEW_COLUMN_USAGE 视图是一个基于来自 SYSOBJECTS、SYSDEPENDS 和 SYSCOLUMNS 系统表信息的虚拟表，其中为用在视图定义的每一列都包括一行信息。当执行如下查询时：

```
SELECT * FROM INFORMATION_SCHEMA.VIEW_COLUMN_USAGE
```

DBMS 将返回如表 19.17 所示的关于在视图定义中所使用的当前数据库中的每一列的信息。

表 19.17 信息架构的 VIEW_COLUMN_USAGE 视图中的列

列名	数据类型	描述
VIEW_CATALOG	NVARCHAR(128)	定义使用列的视图所在的数据库名
VIEW_SCHEMA	NVARCHAR(128)	拥有使用列的视图的用户的 ID
VIEW_NAME	SYSNAME	使用列的视图的名称
TABLE_CATALOG	NVARCHAR(128)	定义包含列的表所在的数据库名
TABLE_SCHEMA	NVARCHAR(128)	拥有包含列的表的用户的 ID
TABLE_NAME	SYSNAME	包含列的表的名称
COLUMN_NAME	SYSNAME	在视图中使用的列的名称

为了显示在当前数据库的任何视图中所使用的列（包括在没有访问权限的视图中使用的列）中的信息架构的 VIEW_COLUMN_USAGE 视图的信息，可执行如下 SELECT 语句：

```
SELECT db_name() 'View Database/Catalog',
user_name(viewobj.UID) 'View Owner',
viewobj.name 'View Name',
db_name() 'Table Database/Catalog',
user_name(tableobj.UID) 'Table Owner',
tableobj.name 'Table Name',
syscolumns.name 'Column Name'
FROM sysobjects viewobj, sysobjects tableobj, sysdepends,
syscolumns
WHERE viewobj.xtype = 'V'
AND sysdepends.ID = viewobj.ID
AND sysdepends.depID = tableobj.ID
AND tableobj.ID = syscolumns.ID
AND sysdepends.depnumber = syscolumns.colID
```

技巧 385 理解信息架构的 VIEW_TABLE_USAGE 视图

VIEW_TABLE_USAGE 视图是一个基于来自 SYSOBJECTS 和 SYSDEPENDS 系统表信息的虚拟表，其中为视图定义所用的每个表都包括一行信息。当执行如下查询时：

```
SELECT * FROM INFORMATION_SCHEMA.VIEW_TABLE_USAGE
```

DBMS 将返回如表 19.18 所示的关于在视图定义中所使用的当前数据库中的每个表的信息而与服务器连接的登录 ID 对当前数据库至少有一种访问权限。

表 19.18 信息架构的 VIEW_TABLE_USAGE 视图中的列

列名	数据类型	描述
VIEW_CATALOG	NVARCHAR(128)	定义基于表的视图所在的数据库名
VIEW_SCHEMA	NVARCHAR(128)	拥有基于表的视图的用户的 ID
VIEW_NAME	SYSNAME	基于表的视图的名称
TABLE_CATALOG	NVARCHAR(128)	定义表所在的数据库名
TABLE_SCHEMA	NVARCHAR(128)	拥有表的用户的 ID
TABLE_NAME	SYSNAME	表的名称

为了显示在当前数据库的任何视图中所使用的表（包括在没有访问权限的视图中使用的列）上的信息架构的 VIEW_TABLE_USAGE 视图的信息，可执行如下 SELECT 语句：

```
SELECT db_name() 'View Database/Catalog',
       user_name(viewobj.UID) 'View Owner',
       viewobj.name 'View Name',
       db_name() 'Table Database/Catalog',
       user_name(tableobj.UID) 'Table Owner',
       tableobj.name 'Table Name'
FROM sysobjects viewobj, sysobjects tableobj, sysdepends
WHERE viewobj.xtype = 'V'
AND sysdepends.ID = viewobj.ID
AND sysdepends.depID = tableobj.ID
```

技巧 386 理解信息架构的 ROUTINE_COLUMNS 视图

ROUTINE_COLUMNS 视图是一个基于来自 SYSOBJECTS、SYSCOLUMNS 和 MASTER.DBO.SYSCCHARSETS 系统表信息的虚拟表，其中为表值函数返回的每一列都包括一行信息（表值函数包括返回数据库文件（包括日志文件）的 I/O 统计数据的 FN_VIRTUALFILESTATS 和列出了受 MS-SQL Server 支持的每个排序规则[排序次序]的 FN_HELPCOLLATIONS）。当执行如下查询时：

```
SELECT * FROM INFORMATION_SCHEMA.ROUTINE_COLUMNS
```

DBMS 将返回如表 19.19 所示的关于由表值函数返回的当前数据库中的列的信息，这些信息对与服务器连接的登录 ID 是可用的。

表 19.19 信息架构的 ROUTINE_COLUMNS 视图中的列

列名/数据类型	描述
TABLE_CATALOG NVARCHAR(128)	包含表值函数的数据库名
TABLE_SCHEMA NVARCHAR(128)	拥有表值函数的用户的 ID
TABLE_NAME NVARCHAR(128)	表值函数的名称
COLUMN_NAME NVARCHAR(128)	由表值函数返回的列的名称
ORDINAL_POSITION SMALLINT	表值函数内的惟一标识号码。ORDINAL_POSITION 在每个函数内部而不跨越所有函数时是惟一的

续表

列名/数据类型	描述
COLUMN_DEFAULT NVARCHAR(4000)	列的默认值
IS_NULLABLE VARCHAR(3)	如果列被允许保存 NULL 值，则为 YES；否则为 NO
DATA_TYPE NVARCHAR(128)	列的 SQL 数据类型
CHARACTER_MAXIMUM_LENGTH SMALLINT	具有二进制、字符、TEXT 或 IMAGE 数据的列的最大字符长度；否则为 NULL
CHARACTER_OCTET_LENGTH SMALLINT	具有二进制、字符、TEXT 或 IMAGE 数据的列的最大字节长度；否则为 NULL
NUMERIC_PRECISION TINYINT	具有数字型数据的列的精度；否则为 NULL
NUMERIC_PRECISION_RADIX SMALLINT	具有数字型数据的列的精度基数；否则为 NULL
NUMERIC_SCALE TINYINT	具有数字型数据的列的大小；否则为 NULL
DATETIME_PRECISION SMALLINT	具有 DATETIME 和 INTEGER 数据的列的分型代码
CHARACTER_SET_CATALOG VARCHAR(6)	总是为 MASTER
CHARACTER_SET_SCHEMA VARCHAR(3)	总是为 DBO
CHARACTER_SET_NAME NVARCHAR(128)	具有 TEXT 或字符类型数据的列的字符集的名称；否则为 NULL
COLLATION_CATALOG VARCHAR(6)	总是为 MASTER
COLLATION_SCHEMA VARCHAR(3)	总是为 DBO
COLLATION_NAME NVARCHAR(128)	TEXT 或字符类型数据的列的排序次序的名称
DOMAIN_CATALOG NVARCHAR(128)	如果列包含用户定义的数据类型的话，则为创建用户定义的数据类型所在的数据库名；否则为 NULL
DOMAIN_SCHEMA NVARCHAR(128)	为包含用户定义的数据类型的列创建用户定义的数据类型的用户的 ID
DOMAIN_NAME NVARCHAR(128)	如果列包含用户定义的数据类型，则为用户定义的数据类型的名称；否则为 NULL

为了查看定义信息架构的 ROUTINE_COLUMNS 视图的 Transact-SQL 代码，可执行在技巧 388 “用 MS-SQL Server Enterprise Manager 查看信息架构视图的内容”中概述的过程。

技巧 387 理解信息架构的 VIEWS 视图

VIEWS 视图是一个基于来自 SYSOBJECTS 和 SYSCOMMENTS 系统表信息的虚拟表，其中为当前数据库中的每个视图都包括一行数据。当执行如下查询时：

```
SELECT * FROM INFORMATION_SCHEMA.VIEWS
```

DBMS 将返回如表 19.20 所示的对与服务器连接的登录 ID 可访问的当前数据库中的每个视图的有关信息。

表 19.20 信息架构的 VIEWS 视图中的列

列名/数据类型	描述
TABLE_CATALOG NVARCHAR(128)	定义视图所在的数据库名
TABLE_SCHEMA NVARCHAR(128)	拥有视图的账号的用户 ID
TABLE_NAME NVARCHAR(128)	视图的名称
VIEW_DEFINITION NVARCHAR(4000)	如果视图的定义的文本长度少于或等于 4000 字符, 则为视图的定义的文本; 否则为 NULL
CHECK_OPTION VARCHAR(7)	如果用来创建视图的语句包括 WITH CHECK OPTION, 则为 CASCADE; 否则为 NULL
IS_UPDATABLE VARCHAR(2)	总是为 NO

为了显示当前数据库中所有视图（包括没有访问权限的视图）的信息架构的 VIEWS 视图的信息, 可执行如下 SELECT 语句:

```
SELECT db_name() 'Database/Catalog Name',
user_name(sysobjects.UID) 'View Owner',
sysobjects.name 'View Name',
CASE WHEN EXISTS
(SELECT * FROM syscomments
WHERE syscomments.ID = sysobjects.ID
AND syscomments.colID > 1) THEN
CONVERT(NVARCHAR(4000), NULL)
ELSE syscomments.text
END 'View Definition',
CASE WHEN EXISTS
(SELECT * FROM syscomments
WHERE syscomments.ID = sysobjects.ID
AND CHARINDEX('WITH CHECK OPTION',
UPPER(syscomments.text)) > 0 THEN
'CASCADE'
ELSE 'NONE'
END 'Check Option',
'NO' 'Is Updateable'
FROM sysobjects, syscomments
WHERE sysobjects.xtype = 'V'
AND sysobjects.ID = syscomments.ID
AND syscomments.colID = 1
```

技巧 388 用 MS-SQL Server Enterprise Manager 查看信息架构视图的内容

正如读者在技巧 387 “理解信息架构的 VIEWS 视图” 末尾的示例代码中学过的, MS-SQL Server 把视图的定义（返回视图的结果集的 SELECT 语句）存储在表的 TEXT 列中。所以, 可以用查询（如技巧 387 末尾所示的代码）显示视图的定义。或者, 也可以按下列步骤用 MS-SQL Server Enterprise Manager 查看视图的定义:

1. 在 Start 菜单上单击, Windows 将显示 Start 菜单。
2. 把鼠标指针移到 Start 菜单的 Programs 上, 选择 Microsoft SQL Server 2000 选项, 并在

Enterprise Manager 上单击。Windows 将在 SQL Server Enterprise Manager 应用程序窗口内启动 Enterprise Manager。

3. 在 SQL Server Group 左侧的加号 (+) 上单击显示网络上可用的 MS-SQL Server 的列表。

4. 在有要显示（也可能要修改）其定义的视图的 SQL 服务器左侧的加号 (+) 上单击。

Enterprise Manager 将显示所选 SQL Server 的 Databases、Data Transformation、Management、Security 和 Support Services 文件夹。

5. 在 Databases 文件夹左侧的加号 (+) 上单击显示 SQL 服务器上当前数据库的列表，并随后单击含有要显示其定义的视图的数据库左侧的加号 (+)。对当前项目而言，就是在 Master 文件夹左侧的加号 (+) 上单击。

6. 要显示在技巧 3 “理解关系数据库模型” 扩充的数据库中视图的列表，可在 Views 图标上单击。Enterprise Manager 将用其右窗格显示数据库中的视图的列表，如图 19.1 所示。

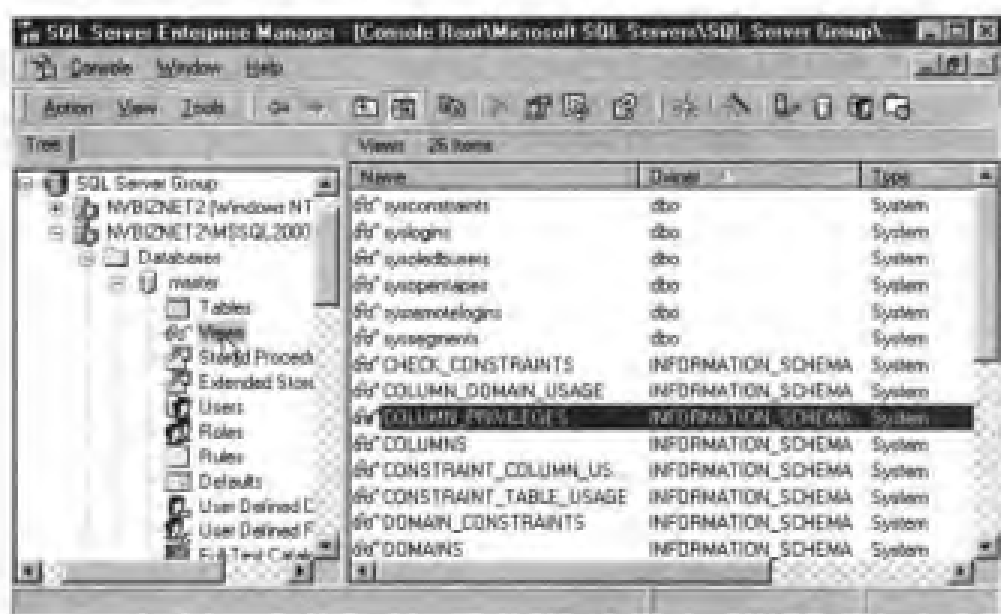


图 19.1 MS-SQL Server Enterprise Manager 列出的 MASTER 数据库中的视图

7. 在要显示其定义的视图上双击。对当前项目而言，就是在 COLUMN_PRIVILEGES（位于右侧面板中）上双击。Enterprise Manager 将在 View Properties 对话框中显示视图的定义的文本，如图 19.2 所示。

8. 要退出 View Properties 对话框（并返回 Enterprise Manager 的主窗口），可按键盘上的 Esc 键（或者在 View Properties 对话框底部的 OK 按钮或 Cancel 按钮上单击）。

除了显示视图的文本，也可以用 Enterprise Manager 编辑视图的定义。要编辑视图，可在前面过程的第 7 步后在 View Properties 对话框的 Text 域中作出所要作的改变。然后，通过在 OK 按钮上单击而不是按 Esc 键（或在 Cancel 按钮上单击）来退出 View Properties 对话框。Enterprise Manager 将检查视图定义的句法并保存更新后的视图定义（如果句法是正确的）（如果句法不正确，Enterprise Manager 将显示一个给出错误源的消息，这样就可以进行必要的更正）。

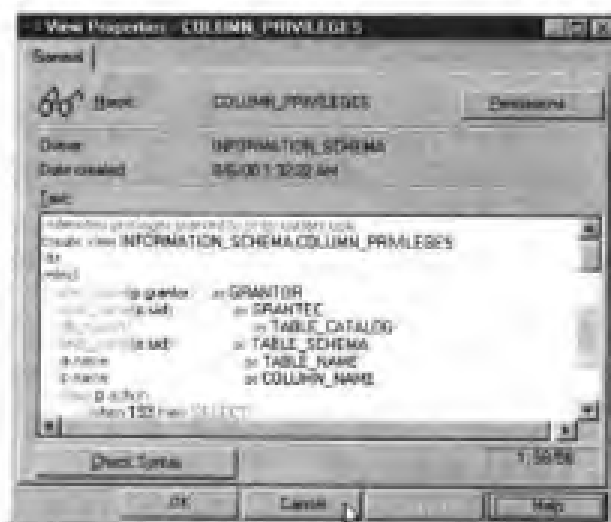


图 19.2 View Properties 对话框中

技巧 389 理解 MS-SQL Server 系统数据库表

当需要由 MS-SQL Server 管理的数据库的信息时，应当使用一个或多个信息架构视图（在技巧 367~技巧 387 讨论过）、系统存储过程、Transact-SQL 语句和函数。通过不直接操作系统表，避免了当 MS-SQL Server 从一个版本升级到下一个版本时，如果 Microsoft 改变系统表的结构或者它的一列或多列的意义时，不得不重写 SQL 代码并防止引入错误。然而，MS-SQL Server 确实允许直接操作用来管理表、索引、外键、用户、权限等的系统表。

表 19.21 描述了 MS-SQL Server 放于在 MS-SQL Server 上创建的每个数据库中的每个系统表。

表 19.21 MS-SQL Server 维护的它所管理的每个数据库中的系统数据库表

表名	描述
SYSCOLUMNS	为数据库中每个表和视图的每一列都包含一行。每行的列都给出目标列的属性（如名称、位置和数据类型）并描述其行为
SYSCOMMENTS	为数据库中的每个 CHECK 约束、DEFAULT 约束、规则、存储过程、触发器和视图都包含一行或多行。表的 TEXT 列包含定义约束、规则、存储过程、触发器和视图的 SQL 或 Transact-SQL 语句。因为定义的大小可以多达 4M，而 TEXT 列只能保存 4000 字节（字符），所以单个对象在 SYSCOMMENTS 表内可能用多于一行描述
SYSCONSTRAINTS	为数据库中的每个约束都包含一个描述名称、ID 和类型的行。表的 ID 列给出拥有约束的表的名称
SYSDEPENDS	描述包含在定义内的对象（存储过程、视图和触发器）与对象（存储过程、存储过程和视图）之间的依赖性
SYSFILEGROUPS	为数据库中的每个文件组都包含一行
SYSFILES	为数据库中与文件组连接的以及用来储存对象的每个物理磁盘文件都包含一行。每行的列描述如文件在磁盘上的物理位置、其文件名、它的当前大小、最大尺寸及其增长率之类的内容
SYSFOREIGNKEYS	为表定义中的每个 FOREIGN KEY 约束都包含一行
SYSFULLTEXTCATALOGS	为在数据库中创建的每个全文索引都包含一行

续表

表名	描述
SYSINDEXES	数据库中的每个索引、表、和 BLOB 都包含一行。每行中的列都描述加对句的第一页数据的位置、行的大小、行数、表的名称、键列的名称之类的内容
SYSINDEXKEYS	为数据库中每个索引的每一列都包含一个描述索引中列的表 ID、索引 ID、列 ID 和次序位置的行
SYSMEMBERS	为数据库中每个角色和（Windows）组的每个专员都包含一个给出了用户 ID、（Windows）组 ID 和角色 ID 的行
SYSOBJECTS	为数据库中的每个对象（约束、默认、索引、日志、规则、存储过程、表等）都包含一行
SYSPERMISSIONS	为数据库对象上授予或拒绝每个用户或角色的每个权限都包含一个描述对象 ID、授权者、被授权者的行
SYSPROTECTS	为通过执行 GRANT 或 DENY 语句在数据库对象上授予或拒绝的每个权限都包含一行。每行中的列都描述被授予或拒绝的权限的类型、用户或角色的 ID、对象 ID、授权者、以及权限（或拒绝）所应用的列的位置图（对 SELECT 和 UPDATE 语句而言）
SYSREFERENCES	为数据库中每个表的每个 FOREIGN KEY 约束都包含一行
SYSTYPES	为数据库中每个系统提供的或用户定义的数据类型都包含一行
SYSUSERS	为数据库中的每个 Windows 用户、Windows 组、MS-SQL Server 用户和 MS-SQL Server 角色都包含一行

技巧 390 定义数据库的物理位置

当使用技巧 24“使用 CREATE DATABASE 语句创建 MS-SQL Server 数据库和事务处理日志”中的 CREATE DATABASE 语句时，为文件指定了 DBMS 将用来存放 FILENAME 子句中的所有数据库对象的物理位置，如：

```
CREATE DATABASE SQLTips
ON (NAME = SQLTips_data,
FILENAME = 'c:\mssql7\data\SQLTips_data.mdf',
SIZE = 10, FILEGROWTH = 1MB)
LOG ON (NAME = 'SQLTips_log',
FILENAME = 'c:\mssql7\data\SQLTips_log.ldf',
SIZE = 3, FILEGROWTH = 1MB)
```

所以，在当前示例中 DBMS 将把在名为 SQLTIPS_DATA.MDF 的文件中创建的每个表都放在服务器的 C 盘的\MSSQL7\DATA\文件夹里（必要时服务器将扩充 SQLTIPS_DATA.MDF 的大小从而超出 10MB 的初始大小以应对所创建和填充的表中的数据的存储空间的需求，一次扩充 1MB）。

为把单个的大数据库（MDF）文件分割成多个较小的文件或允许它扩充磁盘的容量，MS-SQL Server 允许在文件组中定义不止一个物理数据文件。必要的话 MS-SQL Server 将随后扩充每个文件组中的单个物理文件以储存表中的数据并把要存放表（及其数据）的文件组指定为用来创建表的 CREATE TABLE 语句的一部分。

例如, 如下 CREATE DATABASE 语句:

```
CREATE DATABASE company_db
ON PRIMARY (NAME = company_data,
FILENAME = 'c:\msql\data\co_data.mdf',
SIZE = 10, FILEGROWTH = 1MB),
FILEGROUP marketing (NAME = marketing_data,
FILENAME = 'c:\msql\data\mkt_data.mdf',
SIZE = 10, FILEGROWTH = 1MB),
FILEGROUP office (NAME = office_data,
FILENAME = 'd:\msql\data\ofce_data.mdf',
SIZE = 10, FILEGROWTH = 1MB)
LOG ON (NAME = 'co_db_log',
FILENAME = 'c:\msql\data\db_log.ldf',
SIZE = 3, FILEGROWTH = 1MB)
```

将创建3个文件组 PRIMARY、MARKETING 和 OFFICE。可以随后用如下 CREATE TABLE 语句:

```
CREATE TABLE call_history
(call_time DATETIME,
hangup_time DATETIME,
called_by VARCHAR(3),
disposition VARCHAR(4)) ON marketing
```

把 CALL_HISTORY 表放在 C 盘的 MARKETING 文件组中, 并用下面的 CREATE TABLE 语句:

```
CREATE TABLE orders
(customer_ID INTEGER,
order_date DATETIME,
order_total MONEY,
date_shipped DATETIME) ON office
```

把 ORDERS 表放在 D 盘的 OFFICE 文件组中。

数据库中跨越两个或多个驱动器的分布式表允许利用并列磁盘 I/O 操作。用一个磁盘上的 CALL_HISTORY 表和另一个磁盘上的 ORDERS 表, DBMS 可以在进行向 ORDERS 表添加一行的操作的同时进行向 CALL_HISTORY 表添加一行的操作。如果两个表都在相同的驱动器上, 系统必须在启动其他 I/O 操作前完成一个 I/O 操作。

注意: 每个 DBMS 产品都有自己的指定数据库文件和数据库内的表的物理位置的方法。同样, 请查看系统手册了解自己的 DBMS 所要求的语句句法。现在要理解的重要事情是所有的 DBMS 产品都允许 DBA 管理数据库文件的物理位置并可选择 DBMS 用来存放单个数据库表的文件。

技巧 391 向已有数据库添加文件和文件组

在技巧 24 “使用 CREATE DATABASE 语句创建 MS-SQL Server 数据库和事务处理日志”中, 已经学过如何用 CREATE DATABASE 语句创建有多个文件组和多个文件分布于两个物理磁盘驱动器上的 MS-SQL Server 数据库。如果正在管理常规业务环境中的 SQL 服务器, 随着公司不断收集和保留市场、销售和客户服务数据, 数据库的存储需求将最终超过初始存储能力。而

且，技术进步和本部门可用的额外资金使用新的容量更大、速度更快的联机存储解决方案成为可能。幸运的是，MS-SQL Server 提供了 Transact-SQL ALTER DATABASE 命令，该命令允许向已经在使用的驱动器添加额外的驱动器来保存 DBMS 文件。

例如，如果已经创建了技巧 24 中的 SQLTips 数据库，并且后来想要添加一个逻辑驱动器符为 D 的新硬盘（或 RAID 磁盘阵列），可用如下 ALTER DATABASE 语句：

```
ALTER DATABASE SQLTips
ADD FILE (NAME = SQLTips_data2,
FILENAME = 'd:\mssql\data\SQLTips_data2.mdf',
SIZE = 30, FILEGROWTH = 5MB)
ALTER DATABASE SQLTips
ADD LOG FILE (NAME = 'SQLTips_log2',
FILENAME = 'd:\mssql\data\SQLTips_log2.ldf',
SIZE = 3, FILEGROWTH = 1MB)
```

从而允许数据库及其日志文件在 D 盘上增长（在执行添加新驱动器的 ALTER DATABASE 语句后，DBMS 将自动开始使用新的（磁盘）文件空间并在必要时跨越两个磁盘文件分割单个表中的数据）。

除了向数据库添加新文件，也可以用 ALTER DATABASE 语句添加新的文件组并向已有文件组添加文件。例如，假设已经创建了技巧 390“定义数据库的物理位置”中的 COMPANY_DB 数据库。如果执行如下 ALTER DATABASE 语句：

```
ALTER DATABASE company_db 新文件组
ADD FILE (NAME = marketing_data2,
FILENAME = 'e:\mssql\data\mkt_data.mdf',
SIZE = 10, FILEGROWTH = 1MB)
TO FILEGROUP marketing
ALTER DATABASE company_db ADD FILEGROUP multimedia
ALTER DATABASE company_db
ADD FILE (NAME = blobs,
FILENAME = 'i:\mssql\data/blob_data.mdf',
SIZE = 100, FILEGROWTH = 25MB)
TO FILEGROUP multimedia
```

则 MS-SQL Server 将随后开始使用第二个文件（在本例中是 MARKETING_DATA2）存储在 MARKETING 文件组中创建的表。而且，可以创建 TEXT 和 IMAGE（BLOB）数据在 I 驱动器上的 MULTIMEDIA 中的表。

注意：通过把含 BLOB 数据的表放在单独的驱动器上，可以减少当用户执行返回少量数据的查询以及不得不等待硬件返回 BLOB 中的正在被其他用户提取的数据时发生的瓶颈。请记住，BLOB（TEXT 和 IMAGE）数据可以特别大——几兆字节或几吉字节长。所以，BLOB 数据能比典型查询的结果集进行长得多的提取操作，而典型查询的结果集通常只能提取几千或几万字节。

技巧 392 通过创建联接表视图简化多表查询

在用 SQL 工作一段时间后，编写多表联接将变得很自然，因为几乎所有的查询都包括从两个或多个表提取和关联数据。当发现今后还需要几个表的相同数据组合时，创建和使用联接

表视图比重复地重新输入相同的复杂查询要方便得多。而且,许多不熟悉 SQL 和没有时间深入学习 SQL 的管理人员只知道如何编写简单的单表查询。同样,将经常接到非正规处理数据库的请求,以便将信息管理需求放于单一的表中而不是分布于多个管理员必须联接的相关表之间来得到所需的摘要报告。

胜于非正规处理数据库并冒着数据库由于异常修改而崩溃的危险,可创建视图或虚拟表。通过使用视图组合多个表的数据,管理员将会拥有含有几个表的数据的单个(虽然是虚拟的)表而且能够用单表 SELECT 语句从数据库得到所需的数据。

例如,假设销售经理需要为销售部内的每个电话代理人生成产品报告。经理将需要 EMPLOYEES 表中的雇员姓名、APPOINTMENTS 表中的任命数或订货量、CUSTOMERS 中已经发送的销售量和销售额。在创建联接这 3 个表(EMPLOYEES、SALES 和 CUSTOMERS)的如下视图之后:

```
CREATE VIEW VW_Productivity (ID, Phone_Rep_Name)
Appointments, Sales, Sales_Volume, Deliveries,
Total_Revenue) AS
SELECT emp_ID, f_name + ' ' + i_name,
(SELECT COUNT(*) FROM appointments
WHERE phone_rep = emp_ID),
(SELECT COUNT(*) FROM appointments
WHERE phone_rep = emp_ID AND disposition = 'Sold')
COALESCE((SELECT SUM(order_total)
FROM appointments WHERE phone_rep = emp_ID),0),
(SELECT COUNT(*)
FROM customers WHERE phone_rep = emp_ID),
COALESCE((SELECT SUM(contract_total)
FROM customers WHERE phone_rep = emp_ID),0)
FROM employees
WHERE department = 'Marketing' AND status = 'A'
```

销售经理就能够执行单表查询:

```
SELECT * FROM vw_productivity ORDER BY total_revenue DESC
```

来获得按每个为公司创造的收入量递减的顺序排列的活动电话代理人的列表。

技巧 393 理解 CREATE VIEW 语句中的 WITH SCHEMABINDING 子句

以一个或多个表中的列为基础创建视图的一个问题是表的所有者可能会改变表的定义或者删除视图中使用的列,或连表一块删除。遗憾的是,当正在删除视图中使用的列或表时,ALTER TABLE 语句和 DROP TABLE 语句都不会生成能让用户知道这些情况的错误或警告。同样,如果执行如下 ALTER TABLE 语句:

```
ALTER TABLE customers DROP COLUMN contract_total
```

DBMS 将从技巧 392“通过创建联接表视图简化多表查询”中使用的 CUSTOMERS 表中删除 CONTRACT_TOTAL 列而不给出警告或错误。然而,当销售经理试图执行 VW_PRODUCTIVITY 视图上的 SELECT 语句时,DBMS 将中止查询并给出错误消息如下:

```
Server: Msg 207, Level 16, State 3, Procedure Productivity,
Line 4.
```

```
Invalid column name 'contract total'.
Server: Msg 4413, Level 16, State 1, Line1
Could not use view or function 'vw_productivity' because of
binding errors.
```

经理最可能做的将是给 DBA 发送一个表示不愉快的电子邮件，而 DBA 将不得不指出是谁以及为什么删除了该列，还要指出修复问题需要做什么。对经常使用的视图，由表的修改引起的错误可能几个月也不会显现。

幸运的是，CREATE VIEW 语句允许使用 WITH SCHEMABINDING 子句防止表的所有者（或其他有适当权限的用户）删除在架构绑定的视图中使用的表，并中止任何影响视图定义的 ALTER TABLE 语句。例如，如果创建如下视图：

```
CREATE VIEW vw1A_sales (ID, Phone_Rep_Name, Appointments,
Sales, Sales_Volume) WITH SCHEMABINDING AS
SELECT emp_ID, f_name + ' ' + l_name,
(SELECT COUNT(*) FROM frank.appointments
WHERE phone_rep = emp_ID),
(SELECT COUNT(*) FROM frank.appointments
WHERE phone_rep = emp_ID AND disposition = 'Sold')
COALESCE((SELECT SUM(order_total)
FROM frank.appointments WHERE phone_rep = emp_ID),0)
FROM linda.employees
WHERE department = 'Marketing' AND status = 'A'
```

则 DBMS 将不允许 FRANK 删除 (DROP) APPOINTMENTS 表或 INNDA 删除 (DROP) EMPLOYEES 表，而不首先删除 VW_SALES 视图或改变视图定义使之不包括要删除的表。而且，没有用户能够在任何表上执行删除视图中使用的列的 ALTER TABLE 语句。

注意：当使用 WITH SCHEMABINDING 子句时，视图中使用的 SELECT 语句必须包括表、视图和用户定义的函数的两部分名称<所有者.对象>。

第 20 章 监测及提高 MS-SQL Server 的性能

技巧 394 理解多处理器 Windows NT 系统上的 MS-SQL Server 多任务与多线程

当考虑运行于 Windows NT 境中的 MS-SQL Server 的可伸缩性和性能时，理解多任务、多线程和并行处理对 MS-SQL Server 性能影响的不同是重要的。

多任务指操作系统在同一时间运行多个程序的能力。虽然一次只有一个程序能拥有对中央处理器（CPU）的控制权，操作系统通过让每个程序使用 CPU 一段时间来造成几个程序同时执行的假象。在程序的 CPU 时间片断的末尾，系统记录程序是在哪里停止的以便在下一次系统给应用程序以对 CPU 的控制权时能从相同的点重新启动程序。然后系统切换到另一个应用程序并让该程序控制使用 CPU 一段时间，如此继续下去。从一个应用程序切换到另一个应用程序只花费操作系统几百毫秒的时间。然而，当分配给程序的 CPU 时间片断之间的时间由于服务器在不断增长的程序数目之间共享 CPU 而增加时，每个程序都将看起来运行得较慢。

每个运行于 NT Server 上的程序称为一个进程。图 20.1 展示了运行于 Windows NT Server 上的 35 个进程（包括 2 个 MS-SQL Server 的实例）中的几个。

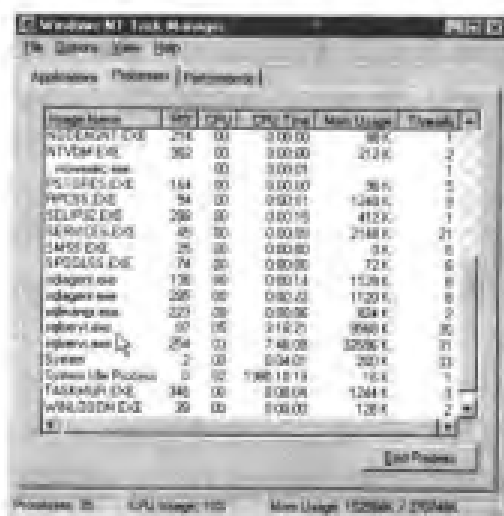


Image Name	PID	CPU	CPU Time	Mem Usage	Threads
smsserver.exe	214	00	0:00.00	48 K	1
ntvdm.exe	362	00	0:00.00	212 K	2
smsservr.exe	400	00	0:00.01	48 K	1
SQLSERVER.EXE	164	00	0:00.00	36 K	3
SQLSERVER.EXE	74	00	0:00.01	1248 K	9
SQLSERVER.EXE	298	00	0:00.00	412 K	1
SQLSERVER.EXE	40	00	0:00.00	2148 K	21
SQLSERVER.EXE	25	00	0:00.00	9 K	8
SQLSERVER.EXE	74	00	0:00.00	72 K	6
sqlservr.exe	138	00	0:00.14	1528 K	8
sqlservr.exe	225	00	0:00.02	1720 K	8
sqlservr.exe	227	00	0:00.00	824 K	2
sqlservr.exe	50	00	0:16.25	9560 K	30
sqlservr.exe	254	00	7:48.08	32536 K	31
System	2	00	0:04.07	20 K	33
System Idle Process	0	00	1:00:18.19	16 K	1
taskmgr.exe	340	00	0:08.04	1244 K	3
winlogon.exe	20	00	0:00.00	128 K	2

图 20.1 Windows NT 任务管理器展示的运行于 NT Server 上的进程的统计数据

操作系统给每个进程分配从 1~31 的优先权（优先权越大，进程等待其 CPU 时间片断的时间越长），并授予具有最高优先权的进程以 CPU 使用权。例如，MS-SQL Server 的每个实例默认得到初始（基本）优先权 7。（运行于基本优先权 7 的进程被说成是运行于普通优先权，因为大多数应用程序是以基本优先权 7 启动的。）在技巧 395“用 MS-SQL Server 的 PRIORITY BOOST 配置选项把服务器线程的优先权从 7 增加到 13”中将学习如何用 PRIORITY BOOST 配置选项把 MS-SQL Server 的基本优先权增加到 13，这将导致 MS-SQL Server 进程以较高优先权运行。

多线程与并行处理的相关之处在于，运行于 NT Server 的每个线程都可能被编写成几个段

以便应用程序中的每个段或线程都能独立地在 CPU 上运行。所以，当 NT 操作系统将一个 CPU 时间片断给予一个多线程的进程时，系统实际上是在线程上次不得不把 CPU 让给另一个进程或线程而暂停之处启动（或重新启动）最高优先权的线程（或进程内的代码段）。例如，MS-SQL Server 的一个实例有 30 个线程，而另一个有 31 个线程。所以，当 Windows NT 把控制交给 MS-SQL Server 的这两个实例中的一个时，它实际上是把 CPU 时间片断给正在等待执行的具有最高优先权的 MS-SQL Server。

MS-SQL Server 的每个实例总是运行几个线程：用来与工作站或其他服务器通信的每个网络协议一个，处理登录请求的一个，另一个与服务器的服务控制管理器通信，其他几个则执行用户发送到 MS-SQL Server 供执行的单个 SQL 和 Transact-SQL 语句以及语句批处理处理。当 NT Server 有多个处理器时具有多线程的多任务是最有利的，因为单个 CPU（或处理器）一次只能执行一个线程。

并行处理指 Windows NT 操作系统将进程和线程分布于单个服务器的多个 CPU 间的能力。所以，有 5 个 CPU 的 Windows NT Server 能同时执行 5 个线程或进程。当用于安装 MS-SQL Server 时，多 CPU 的 NT Server 特别有利，因为 MS-SQL Server 能够用其中一个 CPU 执行数据库 I/O 请求，而同时用另一个 CPU 执行登录以允许用户打开与 SQL 服务器的连接并为另一个用户在第 3 个 CPU 上执行查询等。

技巧 395 用 MS-SQL Server 的 PRIORITY BOOST 配置选项把服务器线程的优先权从 7 增加到 13

正如读者在技巧 394“理解多处理器 Windows NT 系统上的 MS-SQL Server 多任务与多线程”中学习过的，Windows NT 把 1~31 的优先权分配给正等待服务器的一个 CPU 上执行的每个进程和线程。如果有多于一个进程或线程在等待执行，操作系统将把可用的 CPU 时间片断给具有最高优先权的线程。每个 MS-SQL Server 默认为正常的优先权 7。然而，可以用 PRIORITY BOOST 选项把线程的基本优先权增加到 13。因为这样做将给 MS-SQL Server 线程以比服务器正等待执行的其他进程和进程更高的优先权，MS-SQL Server 将倾向于更快地执行查询和其他 DBMS 命令。实际上，在 13 的（高）基本优先权下，NT Server 将倾向于只要线程一准备好执行就执行它（因为这些线程倾向于比其他线程和进程具有更高的优先权）。而且，高优先权的 MS-SQL Server 线程将不能被其他进程中的线程抢先——还是因为它们将倾向于比等待使用 CPU 的其他进程和线程有更高的优先权（当一个线程被迫把对 CPU 的控制权放弃给另一个正在等待的具有更高优先权的线程时，它就被抢先了）。

为了把 MS-SQL Server 线程的基本优先权从 7（正常的）提高到 13（高的），可执行以下步骤：

1. 在 Start 菜单上单击，Windows 将显示 Start 菜单。
2. 把鼠标指针移到 Start 菜单的 Programs 上，选择 Microsoft SQL Server 2000 选项，并在 Enterprise Manager 上单击。Windows 将在 SQL Server Enterprise Manager 应用程序窗口内启动 Enterprise Manager。
3. 在 Microsoft SQL Servers 左侧的加号（+）上单击并随后在 SQL Server Group 左侧的加号（+）上单击显示网络上可用的 MS-SQL Server 的列表。
4. 在要提高其基本优先权的 SQL 服务器名称上右击，并从弹出菜单中选择 Properties 项。

Enterprise Manager 将显示 Server Properties (Configure)对话框。

5. 在 Processor 选项卡上单击。

6. 在 Boost SQL Server Priority on Windows 左侧的复选框上单击, 直到出现选定标记。

7. 在 OK 按钮上单击更新 MS-SQL Server 的配置并返回 Enterprise Manager 主窗口。

在完成第 7 步后, 必须停止并重新启动 MS-SQL Server 以使优先权的提高生效。

注意: 运行具有 PRIORITY BOOST 的 MS-SQL Server 能够大大提高 SQL 服务器的性能。然而, 如果 MS-SQL Server 正在执行要花费长时间才能完成的使用内存较多操作 (如排序), 其他应用程序就不可能有足够高的优先权去抢先 MS-SQL Server 线程。结果, 运行于正常优先权 (优先权 7) 的其他 MS-SQL Server 实例或其他应用程序将会受到相反的影响, 因为这些线程和进程被强迫在得到 CPU 时间片断前等待优先权提高了的线程完成其执行操作。

技巧 396 理解 NT Server 的性能监视器的图表视图

Windows NT Server 包含一个可以用来检查网络上一台或多台计算机 (包括工作站和服务器的性能) 的性能监视器 (Performance Monitor)。性能监视器可以监视的项目 (被应用程序称为对象) 包括处理器 (CPU)、正在执行的程序 (进程)、线程、硬盘、网络资源、Internet 服务和内存。要启动 Windows NT 的性能监视器并生成 MS-SQL Server 统计数据的图形显示, 可执行下列步骤:

1. 在 Windows NT 的 Start 按钮上单击。Windows 将显示 Start 菜单。

2. 把鼠标指针移到 Start 菜单的 Programs 项。选择 Administrative Tools (Common)选项, 并在 Performance Monitor 上单击。Windows 将以如图 20.2 所示的图表视图 (Chart View) 启动性能监视器。

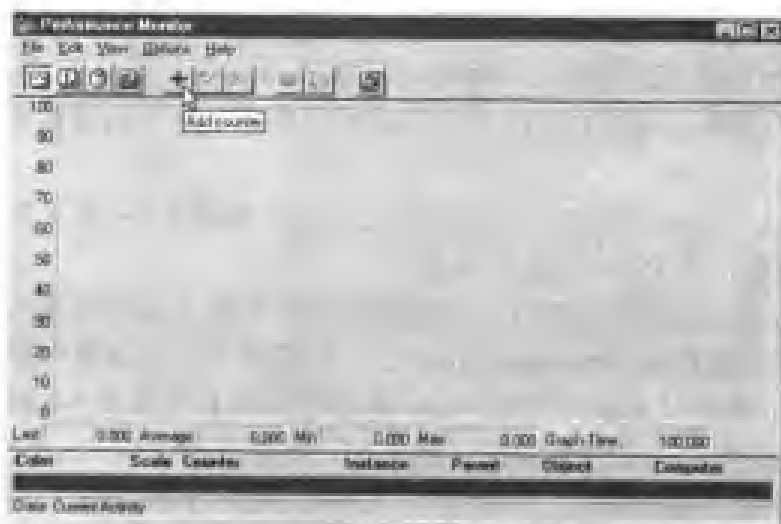


图 20.2 Windows NT 性能监视器的图表视图

3. 为了向图表视图中显示的图表中添加对象 (要监视的项目), 可选择 Edit 菜单的 Add to Chart 选项, 或在 Standard 工具栏的 Add counter 按钮 (带加号[+]的按钮) 上单击。性能监视器将显示 Add to Chart 对话框, 如图 20.3 所示。

4. 在 Computer 域中输入正在运行希望监视的 SQL Server 的 NT Server 的网络路径。如果是在正在运行要监视的 SQL Server 的 NT Server 上启动性能监视器的, 可以接受默认设置。如

果不知道服务器的路径，在 Computer 域右侧的 Search（搜索）按钮（标有 3 个点）上单击，然后从 Select Computer 对话框中的下拉列表中选择想要的 NT Server。

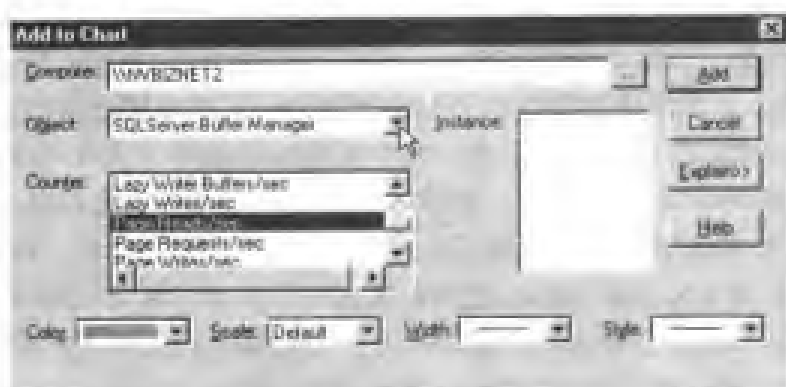


图 20.3 Windows NT 性能监视器的 Add to Chart 对话框

5. 为选择要监视的资源类型，可在 Object 域右侧的下拉列表按钮上单击，然后从下拉列表中选择对象。对当前项目而言，就是从可以监视的对象的下拉列表中选择 SQLServer:Buffer Manager。

注意：MS-SQL Server 的名称可能不同于系统上的 SQLServer。例如，MS-SQL Server 2000 可能被标记为 MSSQL\$<文件夹名>。所以，如果把 MS-SQL Server 2000 安装到 MSSQL2000 文件夹，就要在第 5 步中选择 MSSQL\$MSSQL2000:Buffer Manager。

6. 在 Counter 列表框右侧的滚动条上单击为第 5 步中选择的对象查找要显示的特定计数器值。对当前项目而言，就是在 Counter 列表框中选择 Page Reads/sec。

7. 在 Add 按钮上单击把为第 5 步中选择的对象在第 6 步中所选择的 Counter（计数器）添加到性能监视器图表视图的显示中。

8. 为每个要添加到性能监视器实时图表的对象/计数器重复第 4 步~第 7 步。每次添加新的对象/计数器时，性能监视器将自动改变它将来在图表上画所选择的计数器的统计图的线条的颜色。

9. 为退出 Add to Chart 对话框并返回到所选择的（在第 4 步~第 8 步中）对象/计数器项的性能监视器图表视图，在 Done 按钮上单击。

如果决定要向图表添加额外的项目，可以重复前面从第 3 步开始的过程。相反，要删除对象/计数器，在不再想要在 Performance Monitor 应用程序窗口底部的对象/计数器项目的列表中显示的项目上单击，然后从 Edit 菜单的 Chart 选项选择 Delete，或在 Standard 工具栏的 Delete Selected Counter（删除所选计数器）按钮（标有 X）上单击。

要清除所有的对象/计数器项目，选择 File 菜单的 New Chart 选项。

技巧 397 理解 NT Server 性能监视器的报告视图

在技巧 396 “理解 NT Server 的性能监视器的图表视图”中，已经学过如何用性能监视器以图形显示对象/计数器的信息。当想要查看对象/计数器项目的值随时间的变化或想要比较两个或更多对象/计数器项目的值时，图表工作得很好。然而，有时想要以表格（相对于图表）形式查看性能监视器的统计数据。性能监视器的报告视图（Report View）将显示所选择的对象/计数器项目的数字值。在图表视图中，必须根据图上的线相对于图表纵轴上数据点的高度

的位置来解释项目的值。

如要在报告视图中显示性能监视器的对象/计数器项目，可执行下列步骤：

1. 在 Windows 的 Start 按钮上单击鼠标指针。Windows 将显示 Start 菜单。
2. 把鼠标指针移到 Start 菜单的 Programs 项。选择 Administrative Tools (Common) 选项，并在 Performance Monitor 上单击。Windows 将以技巧 396 的图 20.2 中见过图表视图 (Chart View) 启动性能监视器。
3. 选择 View 菜单上的 Report 选项，或在 Standard 工具栏的 View Report data 按钮 (左起第 4 个按钮) 上单击。
4. 要向报告添加对象 (要监视的项目)，可选择 Edit 菜单的 Add to Report 选项，或在 Standard 工具栏的 Add counter 按钮 (带加号[+]的按钮) 上单击。性能监视器将显示 Add to Report 对话框 (类似于技巧 396 的图 20.3 中见过的 Add to Chart 对话框)。
5. 在 Computer 域中输入正在运行希望监视的 SQL Server 的 NT Server 的网络路径。
6. 为选择要监视的资源类型，可在 Object 域右侧的下拉列表按钮上单击，然后从下拉列表选择一个对象。对当前项目而言，就是选择 SQLServer:Databases。
7. 在 Counter 列表框右侧的滚动条上单击查找要显示的关于第 6 步中所选对象的特定事实 (或计数器)。对当前项目而言，就是在 Counter 列表框中选择选择 Data File(s) Size (KB)。
8. 在 Instance 列表框中列出的数据库中选择要监视其文件大小的数据库。对当前项目而言，就是选择 SQLTips (如果读者的 MS-SQL Server 上没有 SQLTips 数据库，可选择 Instance 列表框中列出的任意一个数据库)。

注意：性能监视器将显示要在其中选择一个属于一个以上对象 (或“实例”) 的计数器的 Instance 列表框中的项目。相反，如果选择如 SQLServer:Buffer Manager 对象的 Cache Size (pages) 计数器那样的对象/计数器项目，性能监视器会给人以实例选择的机会，因为对象/计数器只能属于单个的项目——MS-SQL Server。

9. 在 Add 按钮上单击把第 7 步和第 8 步中选择的关于第 6 步中所选对象的计数器添加到性能监视器的报告视图显示。

10. 为每个想要添加到性能监视器的表格式报告中的对象/计数器/实例重复第 5 到 9 步。对当前项目而言，就是再次执行第 5 步到第 9 步，在第二次执行第 7 步时选择 Log File(s) Size (KB)。

11. 如要退出 Add to Report 对话框并查看所选的 (在第 5 步到第 10 步中) 对象/计数器/实例项目的性能监视器报告视图，在 Done 按钮上单击。性能监视器将显示如图 20.4 所示的表格式报告的。



图 20.4 Windows NT 性能监视器的报告视图

如果决定要向表格式报告添加额外的项目，可以重复前面从第 4 步开始的过程。相反，要删除对象/计数器/实例，在想要删除的项目（在报告体中）上单击，然后从 Edit 菜单的 Report 选项选择 Delete，或在 Standard 工具栏的 Delete Selected Counter 按钮（标有 X）上单击。

如要清除所有的对象/计数器/实例项目，可选择 File 菜单的 New Report Settings 选项。

技巧 398 理解 NT Server 性能监视器的警报视图

技巧 396 “理解 NT Server 的性能监视器的图表视图”和技巧 397 “理解 NT Server 性能监视器的报告视图”中，展示了如何用性能监视器显示正在进行的操作的 MS-SQL Server 数据。然而，还有某些所希望的条件从来不会或很少出现。不用观察对应于一种或多种这样的情况发生的尖峰或陷入计数器值的图形或表格式显示，可以告诉性能监视器只有当特定事件发生时才显示警报。

为了使只要对象/计数器项目的值或对象/计数器/实例项目的值低于或超过特定的阈值，性能监视器就显示警报，可执行下列步骤：

1. 在 Windows 的 Start 按钮上单击。Windows 将显示 Start 菜单。
2. 把鼠标指针移到 Start 菜单的 Programs 项。选择 Administrative Tools (Common) 选项，并在 Performance Monitor 上单击。Windows 将以技巧 396 中的图 20.2 中的图表视图（Chart View）启动性能监视器。
3. 选择 View 菜单的 Alert 选项，或在 Standard 工具栏的 View the Alerts 按钮（左起第二个按钮）上单击。
4. 要向警报报告添加对象（要监视的项目），可选择 Edit 菜单的 Add to Alert 选项，或在 Standard 工具栏的 Add counter 按钮（带加号[+]的按钮）上单击。性能监视器将显示如图 20.5 所示的 Add to Alert 对话框。

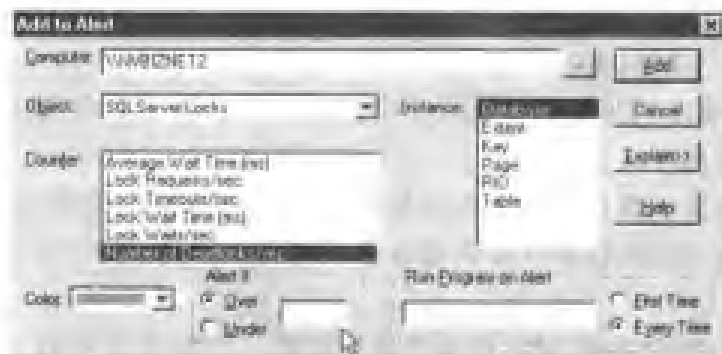


图 20.5 Windows NT 性能监视器的 Add to Alert 对话框

5. 在 Computer 域中输入正在运行希望监视的 SQL Server 的 NT Server 的网络路径。
6. 为选择要监视的资源类型，可在 Object 域右侧的下拉列表按钮上单击，然后从下拉列表表中选择一个对象。对当前项目而言，就是选择 SQLServer:Locks。
7. 在 Counter 列表框右侧的滚动条上单击查找要显示的关于第 6 步中所选对象的特定事实（或计数器）。对当前项目而言，就是在 Counter 列表框中选择 Deadlocks/Sec。
8. 当相同的计数器用于多于一个的项目时，性能监视器将在 Instance 列表框中列出可用的项目（计数器的实例）。同样，只要 Instance 列表框不是空的，就可在其中选择要监视的计数器的实例。对当前项目而言，就是在 Instance 列表框中选择 Database。

9. 进入 Alert If 域, 输入与在第 6 到 9 步所选择的希望性能监视器测试的对象/计数器/实例项目的当前值相对的值。对当前对象而言, 就是在 Alert If 域中输入 0。

10. 如果想要当对象/计数器/实例值的值超过在 Alert If 域中输入的值时, 性能监视器发出警报, 可在 Over 左侧的单选按钮上单击。另外, 还可在 Under 左侧的单选按钮上单击。对当前项目而言, 就是在 Over 左侧的单选按钮上单击。

11. 如果想要性能监视器不论何时接到警报 (通过在警报视图显示中显示一个通知消息) 都执行一个程序, 在 Run Program on Alert 域中输入程序的名称。如果想要性能监视器只在第一次接到警报时执行该程序, 就在 First Time 左侧的单选按钮上单击, 或者如果想要性能监视器每次接到警报时都执行该程序, 就在 Every Time 左侧的单选按钮上单击。

12. 在 Add 按钮上单击, 针对第 7 步和第 8 步中选择的计数器, 把关于第 6 步中选择的对象的警报添加到性能监视器的警报视图。

13. 为每个关于如果值满足在第 9 步和第 10 步指定的警报条件就发出警报 (或者执行程序) 的对象/计数器/实例重复第 5 步和第 12 步。每次添加警报时, 性能监视器都将改变“下一个”警报的颜色, 这样能够容易地把性能监视器警报视图上的某个警报与另一个警报区分开来。

14. 要退出 Add to Alert 对话框, 在 Done 按钮上单击。

性能监视器警报视图将保持空白直到一个对象/计数器/实例值满足警报条件。此后, 性能监视器警将把接到的每个警报添加到显示, 如图 20.6 所示。

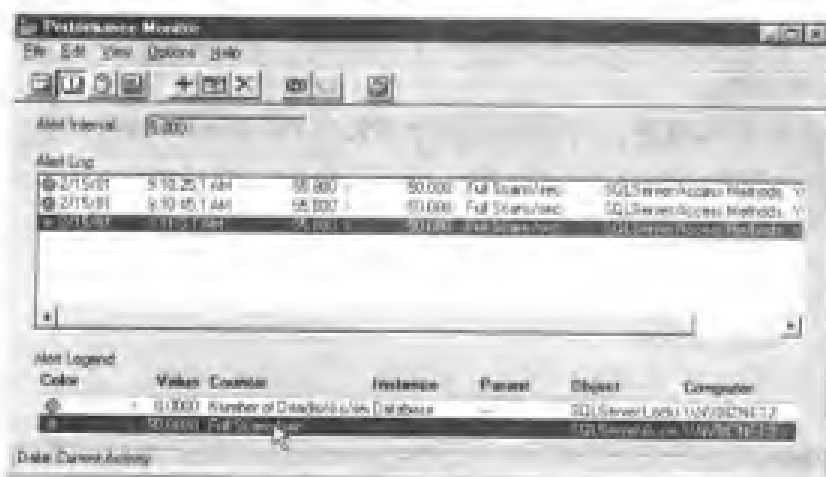


图 20.6 Windows NT 性能监视器警报视图

如果决定要添加额外的警报项目, 可以重复前面从第 4 步开始的过程。相反, 要删除警报, 在 Performance Monitor 应用程序窗口底部的警报列表中不再想要的警报上单击, 然后从 Edit 菜单的 Alert 选项选择 Delete, 或在 Standard 工具栏的 Delete Selected Counter 按钮 (标有 X) 上单击。

如要清除所有的警报, 可选择 File 菜单的 New Alert Settings 选项。

技巧 399 使用 CREATE SCHEMA 语句创建表并授予对此表的访问权限

SQL-89 标准在 SQL 的数据操作语言 (DML) 语句和数据定义语言 (DDL) 语句之间设置了很大的差别。当该标准要求 DBMS 在其正常操作过程中能执行 DML 语句时, 却没有要求关于执行 DML 语句的能力。实际上, SQL-89 标准允许 SQL 数据库有与老式的分层和网络数据库模型。

如果用静态数据库结构实现遵循 SQL-89 的 DBMS，数据库管理员（DBA）将用 DDL 语句创建数据库架构——一种显示包括表、视图、用户和访问权限的数据库结构的数据库映像。DBA 将随后向根据架构中的规范创建数据库的构建器实用工具提交数据库架构。一旦被创建（由构建器实用工具创建），数据库对象及安全架构就不能被改变。DML 语句可以添加、改变、删除和提取数据。然而，要向数据库添加新的表或用户——它要求执行 DDL 语句——DBA 将不得不停止所有对数据库的访问，卸载所有的数据，用 DDL 创建修改过的架构，向构建器实用工具提交新架构，然后重新载入数据库数据。

虽然 SQL-89 标准允许，但实际上没有数据库产品使用静态数据库结构。事实上，后来的 SQL-92 标准包括了 DROP（TABLE、VIEW、USER 等）和 ALTER（TABLE、VIEW、USER 等）语句，实际上都要求遵循 SQL-92 的数据库支持动态数据库对象的定义和修改。然而，用数据库架构创建一组表、视图和权限（在数据库中有效）的概念仍然受到许多 DBMS 产品的支持——不管其快速创建、删除和修改单个数据库的能力。

CREATE SCHEMA 语句允许创建包含表和视图定义的概念性的数据库并允许把这些表和视图上的访问权限授给用户和角色。CREATE SCHEMA 语句的句法如下：

```
CREATE SCHEMA AUTHORIZATION <account ID>
[<table definition(s)>|<view definition(s)>|
<grant statement(s)>]
```

注意：虽然 AUTHORIZATION <账号 ID>是必需的，但 DBMS 分配它所创建的表和视图的所有权时实际上并没有用到它。所有通过执行 CREATE SCHEMA 语句创建的表和视图都由执行语句的用户 ID 所拥有。

为了创建一组数据库和视图，可以执行如下 CREATE SCHEMA 语句：

```
CREATE SCHEMA AUTHORIZATION frank
CREATE VIEW vw_offices AS
SELECT offices.office_ID, manager_ID,
f_name + l_name manager_name
FROM offices, employees
WHERE manager_ID = emp_ID
CREATE TABLE employees
(emp_ID INTEGER,
f_name VARCHAR(15),
l_name VARCHAR(15),
total_sales MONEY,
office_ID SMALLINT)
CREATE TABLE offices
(office_ID SMALLINT,
street_address VARCHAR(30),
manager_ID INTEGER)
GRANT SELECT ON vw_offices TO PUBLIC
GRANT ALL PRIVILEGES ON offices TO sally
GRANT ALL PRIVILEGES ON employees TO sally
```

简而言之，CREATE SCHEMA 没有给 DDL 增加任何实际的功能。然而，CREATE SCHEMA 语句却给出了一个可以用来创建一个或多个表和视图并把访问权限授给这些表和视图的单个

SQL 语句。

由于有视图依赖于其他视图的例外，在 CREATE SCHEMA 语句中创建的对象不必以任何特定的顺序出现。同样，可以以 CREATE SCHEMA 语句中创建的表为基础授予权限或创建视图。而且，在该语句的一部分中创建的视图定义和外键可以引用后来在 CREATE SCHEMA 语句中创建的表中的列。然而，如果视图定义引用另一个视图中的列（相对于表中的列），其列被引用的视图必须在引用它的列的视图之前创建。

正如同任何单个 SQL 语句的情况一样，如果语句的任何部分执行失败，则 DBMS 将撤销 CREATE SCHEMA 语句的工作。所以，如果 CREATE SCHEMA 语句的 CREATE 或 GRANT 语句中的任何一个失败，DBMS 将不创建任何的架构表或视图，也不会授予架构中指定的权限。

技巧 400 建立 NT Server 性能监视器日志以帮助优化 MS-SQL Server

技巧 396 “理解 NT Server 的性能监视器的图表视图”和技巧 397 “理解 NT Server 性能监视器的报告视图”分别展示了如何用性能监视器以图表和表格形式显示对象/计数器的实时值。如果要储存一个或多个性能监视器对象的计数器值以便可以在以后显示并分析计数器值，可使用性能监视器的日志视图（Log View）。性能监视器的日志视图将在设置间隔向磁盘文件写入计数器值。正如读者将在技巧 401 “用 NT Server 性能监视器查看性能日志文件”中学习的，可以告诉性能监视器从日志文件提取对象/计数器值并以图表或图形形式显示。

要使性能监视器记录一个或多个对象的计数器值供以后再次查看，可执行以下步骤：

1. 在 Start 按钮上单击。Windows 将显示 Start 菜单。
2. 把鼠标指针移到 Start 菜单的 Programs 项。选择 Administrative Tools (Common) 选项，并在 Performance Monitor 上单击。Windows 将以技巧 396 的图 20.2 中见过图表视图（Chart View）启动性能监视器。
3. 选择 View 菜单的 Log 选项，或在 Standard 工具栏的 View output Log file status 按钮（左起第 3 个按钮）上单击。
4. 要添加想在性能监视器日志文件中记录其值的对象（计数器集），可选择 Edit 菜单的 Add to Log 选项，或 Add counter 按钮（Standard 工具栏上带加号[+]的按钮）上单击。性能监视器将显示 Add to Log 对话框，如图 20.7 所示。



图 20.7 Windows NT 性能监视器的 Add to Log 对话框

5. 在 Computer 域中输入正在运行希望监视的 SQL Server 的 NT Server 的网络路径（如果不知道服务器的路径，在 Computer 域右侧的 Select（选择）按钮上单击，以便可以从 Select Computer 对话框中的服务器的列表中选择想要的计算机）。

6. 在 Add to Log dialog 对话框的 Objects 部分中的对象列表中要记录的计数器的目录上单击。例如，如果要生成 page reads/sec 值的图表（像技巧 396 中所做的那样），选择包括了 page

reads/sec 计数器的 Buffer Manager 对象。

7. 在 Add 按钮上单击把第 6 步中所选的对象添加到性能监视器将其计数器集的值写入日志文件的对象的列表中。

8. 重复第 6 步和第 7 步直到已经选择了所有想要在以后再次查看其计数器值的对象。然后在 Done 按钮上单击。性能监视器将关闭 Add to Log dialog 对话框并返回到日志视图（如果选择了任何不再想要的对象，就在它上面单击指针，然后可按键盘上的 Delete 键）。

9. 选择 Options 菜单的 Log 选项。性能监视器将显示 Log Options 对话框，如图 20.8 所示。

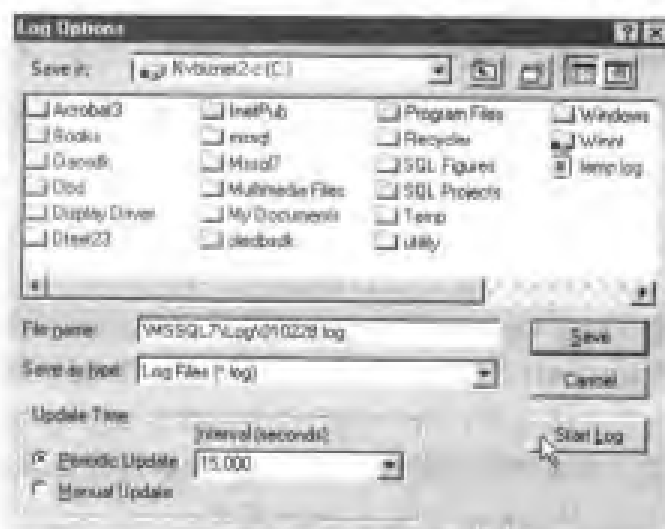


图 20.8 Windows NT 性能监视器的 Log Options 对话框

10. 在 File Name 中输入想要性能监视器在其中存储对象的计数器值的文件的路径名。

11. 在 Periodic Update 域（靠近对话框的左下角）中，以秒为单位输入想要使性能监视器在日志文件中记录值的频率。

12. 在 Start Log 按钮上单击。

当完成第 12 步时，性能监视器将关闭 Log Options 对话框并开始把对象/计数器值（在第 6 步~第 8 步所选择的）以第 11 步中所输入的时间间隔写入到日志文件（在第 10 步中指定的）中。在横线（-）（性能监视器应用程序窗口右上角右起第 3 个按钮）上单击，把应用程序缩小成计算机屏幕底部的任务栏上的一个图标。

在性能监视器（在计算机上作为后台任务运行）已经收集了所希望的时间段的数据后，在屏幕底部任务栏上它的上图标单击鼠标指针显示其应用程序窗口。下一步，选择 Options 菜单的 Log 选项再次显示 Log Options 对话框。要停止应用程序记录计数器值，在 Stop Log 按钮上单击鼠标指针（在对话框的右下角）。

读者将在技巧 401 中学习如何用性能监视器显示日志文件的内容。

技巧 401 用 NT 性能监视器查看性能日志文件

在创建性能监视器日志文件（通过执行技巧 400“建立 NT Server 性能监视器日志文件以帮助优化 MS-SQL Server”中的步骤），就可以用性能监视器以图表（图形）或表格形式显示日志文件的内容。过程很简单。避免使性能监视器从装有 MS-SQL Server 的 NT Server 提取它要直接显示的值，可以告诉程序使用日志文件中的计数器值（就像它们直接取自服务器一样）。

例如, 如果要使用性能监视器以图表形式显示以前记录的计数器值, 请执行下列步骤:

1. 在 Start 按钮上单击。Windows 将显示 Start 菜单。
2. 把鼠标指针移到 Start 菜单的 Programs 项。选择 Administrative Tools (Common) 选项, 并在 Performance Monitor 上单击。Windows 将以图 20.2 中见过的图表视图启动性能监视器(见技巧 396 “理解 NT Server 性能监视器图表视图”)。
3. 选择 View 菜单的 Chart 选项。
4. 选择 Options 菜单的 Data From 选项。性能监视器将显示 Data From 对话框, 如图 20.9 所示。



图 20.9 Windows NT 性能监视器的 Data From 对话框

5. 要使性能监视器收集并显示日志文件中的计数器值, 在 Log File 单选按钮上单击, 并随后在 Data From 底部的域中输入日志文件的路径名(如果在 Current Activity 单选按钮上单击, 性能监视器将直接从服务器收集它要显示的计数器值)。

6. 在 OK 选按钮上单击。

7. 为选择要在图表视图中显示的对象/计数器值, 选择 Edit 菜单的 Add to Chart 选项, 或在 Standard 工具栏的 Add counter 按钮上单击(带加号[+]的按钮)。性能监视器将显示与技巧 396 的图 20.3 所见到的相似的 Add to Chart 对话框。

8. 使用 Computer 域右侧的 Select 按钮(标有 3 个点号[...]), 然后选择要显示其数据值的 NT Server(只有在第 5 步和第 6 步中选择的日志文件中有数据的服务器才会出现在 Search 按钮的服务器下拉列表中)。

9. 要选择想要显示其记录的计数器值的对象, 在 Object 域右侧的下拉列表按钮上单击, 然后从下拉列表中选择对象。只有那些其计数器值被写入到了日志文件中的对象才会出现在可以选择的对象的下拉列表上。

10. 在 Counter 列表框右侧的滚动条上单击为第 9 步中选择的对象查找要显示的特定计数器值。

11. 在 Add 按钮上单击为在第 9 步中选择的对象把第 10 步中所选择的计数器添加到性能监视器的图表视图显示中。

12. 为要添加到性能监视器图表的每个对象/计数器重复第 8 步~第 11 步。每次添加新的对象/计数器时, 性能监视器都将自动改变用来在图表上画所选择的计数器的统计数据的线条颜色。

13. 要退出 Add to Chart 对话框并返回到性能监视器图表视图, 在 Done 按钮上单击。

在完成第 13 步后, 性能监视器将在单个图表上显示整个日志文件的内容。要“放大”在特定的时间段内记录的值, 选择 Edit 菜单的 Time Window 项, 并在 Input Log File Timeframe 对话框中指定感兴趣的时间段。

技巧 402 配置 Windows NT 的应用程序事件日志

在技巧 398 “理解 NT Server 性能监视器的警报视图”中，读者学过如何在对象/计数器值不论何时超过或低于某个限定值时，用性能监视器通过在警报视图中显示一个消息为 MS-SQL Server 操作的潜在问题发出警报。然而，性能监视器不是惟一可以用来监视 MS-SQL Server 运行好坏的工具。Windows NT 包括一个集成的记录工具来维护应用程序、安全和系统操作的日志。通过查阅由 MS-SQL Server 生成的应用程序日志事件，可以知道 SQL Server/Agent 的停止和启动有多频繁，看它遇到的文件（表、日志和索引）错误有多频繁，并审核由各种其他数据库操作如当 SQL 服务器在异常关闭后重启时执行的事务处理回滚或前滚生成的警告和错误消息。

当启动服务器时，Windows NT 会自动启用事件记录服务。为了避免由于全事件日志引起的事件通知丢失，应当在服务器上安装 MS-SQL Server 之后使用事件查看器（Event Viewer）配置事件日志。要使用事件查看器以 KB 为单位设置最大的日志文件大小、在日志中保留事件的时间长度，如果日志文件已记满则事件记录服务是否覆盖已有事件，可执行以下步骤：

1. 在 Start 按钮上单击。Windows 将显示 Start 菜单。
2. 把鼠标指针移到 Start 菜单的 Programs 项。选择 Administrative Tools (Common) 选项，并在 Event Viewer 上单击。
3. 选择 Log 菜单的 Log Settings 选项，事件查看器将显示 Event Log Settings 对话框，如图 20.10 所示。

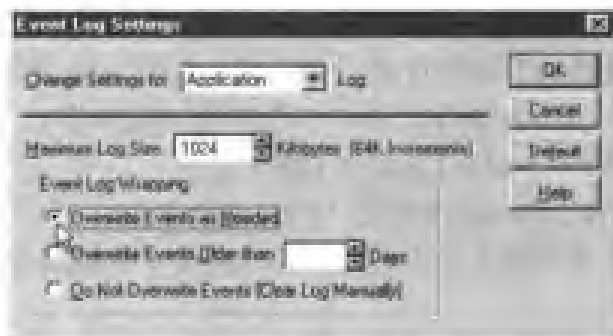


图 20.10 Windows NT 事件查看器的 Event Log Settings 对话框

4. 在 Change Setting For 域（靠近对话框顶部）右侧的下拉列表按钮上单击，从下拉列表中选择 Application。
 5. 在 Maximum Log Size 域中输入事件日志的最大的大小。如果增加日志文件的最大的大小超出了默认的 512Kb，将能在日志文件中保存更多的事件更长的时间以便观察由 MS-SQL Server 报告的事件之间的趋势。
 6. 选择如果日志文件记满，想要事件记录服务进行的动作。要避免丢失新事件，在 Overwrite Events as Needed 单选按钮上单击。通过这样做，当新事件添加到已经记满的日志时事件记录服务将覆盖旧事件（选择这两种设置都可能导致新的事件通知的丢失）。
 7. 在 OK 按钮上单击保存配置的改变并返回事件查看器应用程序窗口。
- 当仍然处于事件查看器中时，还应当继续并设置系统和安全日志的配置设置。为达此目的，从前述过程的第 3 步开始并在第 4 步选择 System。然后，在完成第 5 步和第 6 步后，再次从第 3 步开始重复该过程，在第 3 次执行的第 4 步过程选择 Security。

技巧 403 显示 Windows NT 应用程序事件详情并清除应用程序事件日志

事件查看器允许查看由 Windows NT 服务器的事件记录服务记录的应用程序（系统和安全）日志信息。如图 20.11 所示，事件日志中的每一行都展示了事件发生的日期和时间、事件来源、事件类别、ID 号和启动事务处理的用户的用户名（如果可用的话）。

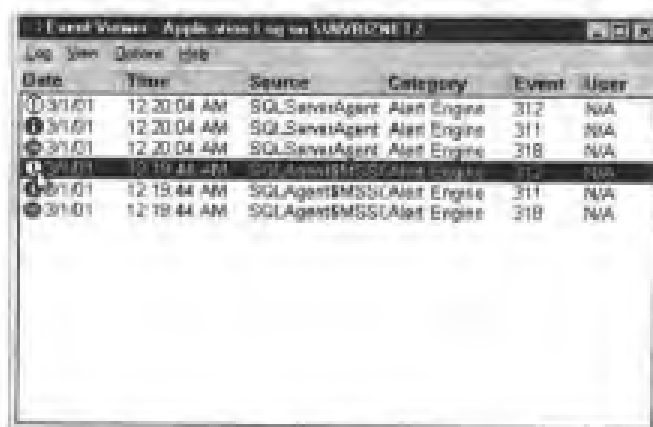


图 20.11 Windows NT 事件查看器的应用程序事件日志

为了显示事件较为详细的信息，在事件上双击（或在事件上单击，然后选择 View 菜单的 Detail 选项）。事件查看器将显示 Event Detail 消息框，如图 20.12 所示。



图 20.12 Windows NT 事件查看器的 Event Detail 消息框

在调用 Event Detail 对话框显示一个事件的详情以后，可以在其 Next 按钮上单击显示下一个事件（向事件列表按钮的底部），或在其 Previous 按钮上单击显示向列表的顶部下一个事件。当查看完事件详情时，在 Close 按钮上单击关闭 Event Detail 对话框并返回到事件视图应用程序窗口。

如果观察到了应用程序日志的事件消息中的趋势，对应用程序的设置或硬件进行了一次纠错，而且现在想要监视服务器重新发生的趋势，就将以空的事件日志启动。要从日志清除

所有事件，选择 Log 菜单的 Clear All Events 选项。事件监视器将显示 Clear Event Log 配置文本框。如果要在清除事件日志前把它写到磁盘文件中，就在 Yes 按钮上单击，在把日志文件的内容保存到磁盘上的另一个文件中之前，在 No 按钮上单击清除事件日志，或者在 Cancel 按钮上单击保持事件日志不变。

技巧 404 用 MS-SQL Server 服务管理器启动 MS-SQL Server

有几种方法可以启动 Windows NT Server 上的 MS-SQL Server。当安装 MS-SQL Server 时，可以在安装屏幕的一个中选择一个选项使不论何时引导操作系统，Windows NT 都自动将 MS-SQL Server 作为一项服务启动。相反，可以选择用 MS-SQL Server 服务管理器（Service Manager）或在 MS-SQL Server 的 BINN 目录中的命令提示符下输入一个命令来手动启动 MS-SQL Server。

如要用 MS-SQL Server 服务管理器启动 MS-SQL Server，可执行下列步骤：

1. 在装有 MS-SQL Server 的 Windows NT Server 的 Start 按钮上单击。Windows 将显示 Start 菜单。
2. 把鼠标指针移到 Start 菜单的 Programs 项上。选择 Microsoft SQL Server 程序组，并在 Service Manager 上单击。Windows NT 将启动会显示一个对话框的 MS-SQL Server 服务管理器，如图 20.13 所示。



图 20.13 MS-SQL Server Service Manager 对话框

3. 如果系统上安装有不只一个 MS-SQL Server，在 Server 域右侧的下拉列表按钮上单击显示已安装的 MS-SQL Server 的列表并选择想要启动的 MS-SQL Server。

4. 在 Services 域右侧的下拉列表按钮上单击显示 MS-SQL 服务管理器控制的服务的列表，并选择 SQL Server。

5. 要启动第 3 步中选择的 MS-SQL Server，在对话框下半部的 Start/Continue 按钮上单击。MS-SQL Server 服务管理器将试图启动 MS-SQL Server。如果启动成功，Server 服务管理器将把对话框中服务器图标上的红色块变成绿色箭头表明有正在运行的服务，即 MS-SQL Server。

注意：如果要每次引导操作系统时 Windows NT Server 都自动启动 MS-SQL Server（服务），在 Auto-start Service When OS Starts 左侧的复选框上单击直到出现选定标记。

6. 要退出 MS-SQL Server 服务管理器，则在对话框右上角的关闭按钮（×）上单击。除了启动 MS-SQL Server，还可以用 MS-SQL Server 服务管理器启动其他 4 个与 SQL 有关的服务。

- Microsoft Search——为基于列的对字符和文本的高级查询建立索引和支持这种查询

的服务

- Microsoft Distributed Transaction Coordinator (MSDTC)——管理事务处理以使应用程序能把运行于相同或不同 Windows NT 服务器上的不同数据源（包括 MS-SQL Server 的多个实例）中的数据组合起来的服务
- MSSQLServerOLAPService——用来维护数据仓库数据的多维数据集并提供对数据仓库数据的快速分析访问的联机分析处理服务。
- SQL Server Agent——运行计划好的如备份、索引更新、为在特定日期和时间执行而安排的存储过程之类的 MS-SQL Server 管理任务的应用程序。

如要启动这些 MS-SQL Server 支持服务中的任何一个，可在前述过程的第 4 步中选择想要启动的服务而不是 SQL Server。

技巧 405 理解如何恢复 MS-SQL Server 数据库

没有人会期待硬件或软件故障。然而，故障仍然一定会发生。幸运的是，由于已经设计并执行全面备份计划，将可在尽可能最短的时间内恢复 SQL 数据库并使数据库联机还原。

通常，恢复 MS-SQL Server 的全部功能包括下列步骤：

1. 在发生硬件故障的情况下，可能不得不自己重新安装 MS-SQL Server（如果确实不得不再次安装 MS-SQL Server，也不必重新创建它的数据库——可以让 MS-SQL Server 的恢复数据库功能为你处理）。
2. 如果数据库被破坏或丢失，就恢复 Master 数据库，它包含了关于备份时 MS-SQL Server 上可用的所有其他数据库和数据库对象的信息。
3. 如果 Master 数据库被破坏或丢失，则恢复 MSDB 数据库，它包含了关于警报、备份、任务和数据库复制计划（如果有的话）的信息。
4. 对每个丢失或被破坏的数据库，恢复其最近的完全备份。
5. 对第 4 步中恢复的每个数据库，按顺序恢复其从最近的完全备份以来的事务处理日志备份。

在启动数据库恢复操作以前，文件服务器上必须安装并运行有功能正常的 MS-SQL Server。所以，如果有必要，请执行 MS-SQL Server 的安装过程。

注意：如果正在进行硬件故障后的数据库数据恢复，只有如果 MS-SQL Server 的程序文件受到某种程度的破坏或被删除才会不得不重新安装 MS-SQL Server。不必为了恢复被破坏或丢失的数据库文件而重新安装 MS-SQL Server。所以，在重新安装 MSSQL Server 软件前，请检查文件服务器的任务列表看 MS-SQL Server 是否已经安装和运行。如果在文件服务器的任务列表中没有显示 MS-SQL Server，或其状态不是 Running，请与网络管理员一起检查以确保要使用的 MS-SQL Server 没有由于其他原因而移到其他的文件服务器上或下线。如果 MS-SQL Server 只是简单的 Stopped，可以用 MS-SQL Server 服务管理器重新启动它。

接着，确保最近的完全备份数据库文件及其后的事务处理日志备份文件可用。如果用基于磁盘的备份计划把 MS-SQL Server 备份到了磁盘，就从必要时就从网络宽度的备份系统的磁盘恢复服务器的备份文件。相反，如果使用了基于磁带的备份计划为 MS-SQL Server 备份，就要确保磁带设备可用，并从存储库中取回所需要的 MSSQL Server 备份磁带。

在 MS-SQL Server 已经设置和运行并且已经拥有了位于磁盘或磁带上的完全数据库备份

文件和事务处理日志备份文件以后，就执行下列步骤从完全备份文件恢复数据库：

1. 通过在 Windows 的 Start 按钮上单击并从 Start 菜单选择 Programs 启动 Enterprise Manager，然后把鼠标指针移到 Microsoft SQL Server 程序组并在 Enterprise Manager 上单击。

2. 为显示 SQL 服务器列表，在 Microsoft SQL Servers 左侧的加号（+）上单击，然后在 SQL Server Group 左侧的加号（+）上单击。

3. 在要恢复其数据库的 SQL 服务器的图标上单击。例如，如果要恢复名为 NVBIZNET2 的 NT 服务器上名为 MSSQL2000 的 SQL 服务器上的 COMPANY_DB 数据库，在 Enterprise Manager 窗格中 NVBIZNET2\MSSQL2000 左侧的图标上单击。Enterprise Manager 将用右窗格显示包含所选服务器上可用的数据库和服务的列表的文件夹的图标。

4. 选择 Tools 菜单的 Restore Database 选项。Enterprise Manager 将显示 Restore Database 对话框，如图 20.14 所示。

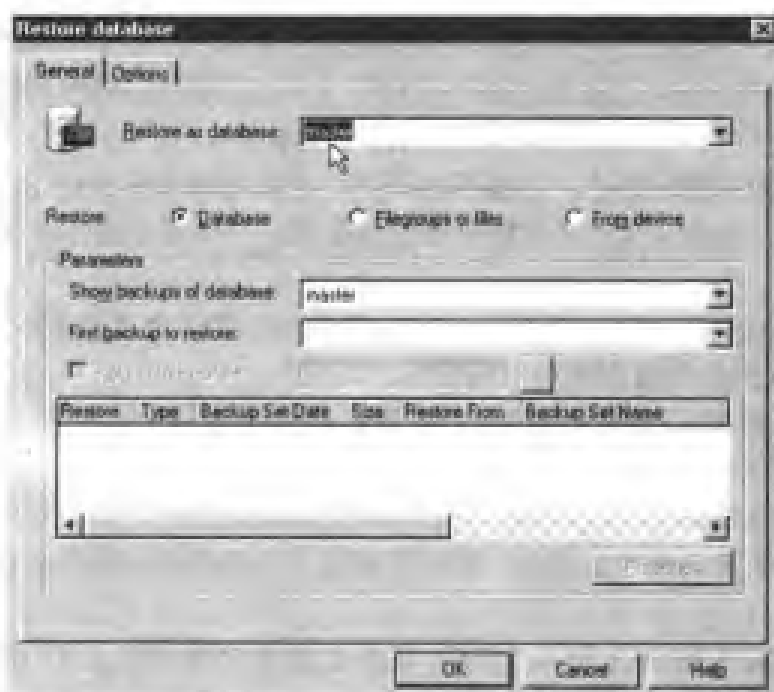


图 20.14 MS-SQL Server Enterprise Manager 的 Restore Database 对话框

5. 进入 Restore As database 域，输入要恢复的数据库的逻辑名。或者，如果计划恢复的数据库已经在 MS-SQL Server 上存在，在 Restore as Database 域右侧的下拉列表按钮上单击，从下拉列表中选择数据库名。

注意：如果正在恢复 MS-SQL Server 上的所有数据库，确保首先恢复 MASTER 数据库。接着，重复当前全备份恢复过程中的步骤恢复 MSDB 数据库，然后安装任何用户创建的数据库。要小心的是不要通过接受 Restore to Database 域中的默认数据库名而把 SQLTips 或 COMPANY_DB 之类的用户创建的数据库作为 MASTER 数据库恢复。

6. 在对话框 Restore 部分的 From Device 单选按钮上单击。Enterprise Manager 将在对话框的 Parameters 部分中显示 Restore from Device 选项，如图 20.15 所示。

7. 在对话框中部的 Device 滚动框右侧的 Select Devices 按钮上单击。Enterprise Manager 将显示 Choose Restore Devices 对话框，如图 20.16 所示。

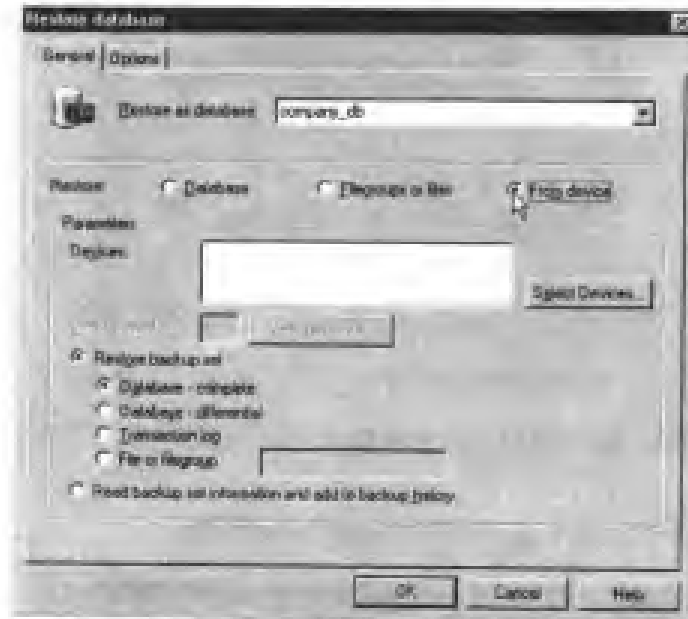


图 20.15 MS-SQL Server Enterprise Manager Restore Database 对话框的 General 选项卡显示了 Restore from Device 选项

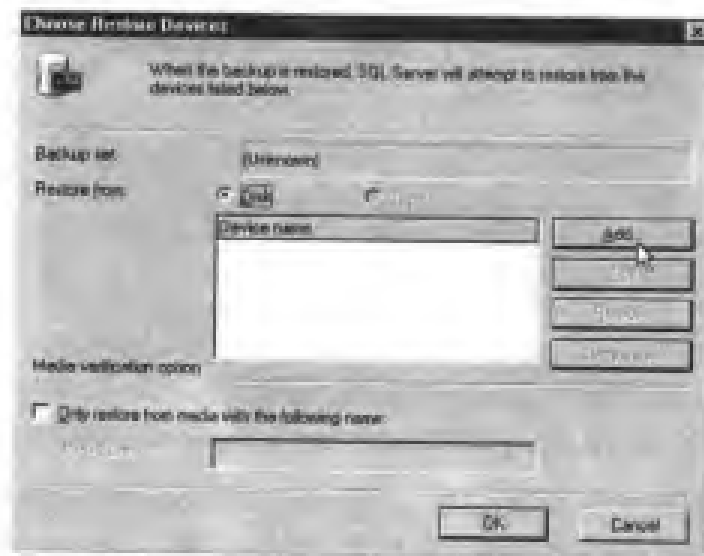


图 20.16 MS-SQL Server Enterprise Manager 的 Choose Restore Database 对话框

8. 在 Disk 单选按钮上单击，然后在对话框中部的 Restore from Set 部分的 Add 按钮上单击。Enterprise Manager 将显示 Choose Restore Destination 对话框，如图 20.17 所示。

9. 在 File Name 单选按钮上单击。然后在 File Name 域中输入 MS-SQL Server 备份文件的全路径名。或者，用 File Name 域右侧的 Browse 按钮导航到有备份文件的文件夹，然后在文件夹名上双击。

10. 在靠近 Choose Restore Destination 对话框右下角的 OK 按钮上单击返回 Choose Restore Devices 对话框。

11. 如果要恢复的备份储存在多个文件中，就在必要时重复第 8 步~第 10 步。

12. 在靠近 Choose Restore Devices 对话框右下角的 OK 按钮上单击返回 Restore Database 对话框。

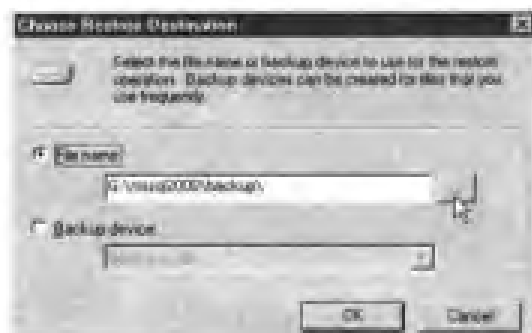


图 20.17 MS-SQL Server Enterprise Manager 的 Choose Restore Destination 对话框

13. 在 Restore Backup Set 单选按钮上单击并随后在对话框下半部的 Database—Complete 单选按钮上单击。
14. 在 Options 选项卡上单击显示如图 20.18 所示的域。

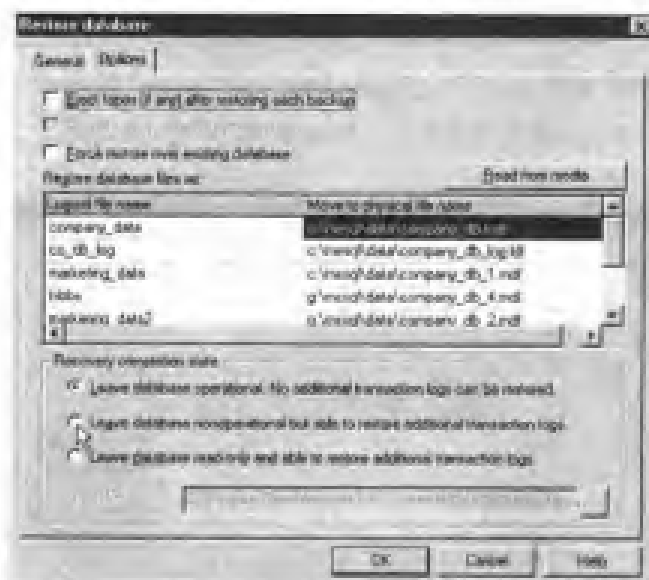


图 20.18 MS-SQL Server Enterprise Manager Restore Database 对话框的 Options 选项卡

15. 重新查看恢复过程将写入 Options 选项卡中心的 Restore Database Files As 滚动框中的物理数据库文件和事务处理日志文件的文件地址。默认情况下，Enterprise Manager 将把备份文件的内容恢复到 Enterprise Manager 执行完全数据库备份过程时所在的文件夹相同的文件夹。如果要改变数据库中任何文件的物理位置，在滚动框 Move to Physical File Name 列中的文件的当前位置上单击，用要使用的位置代替已有的文件位置。

16. 如果有恢复所需要的在恢复完全备份文件以后的事务处理日志文件，在 Leave Database Nonoperational but Able to Restore Additional Transaction Logs 单选按钮上单击。这将防止用户在完成从完全备份文件和完全备份之后随即制作的事务处理日志文件的备份文件中恢复数据库以前登录到数据库。

17. 在靠近 Restore Database 对话框底部中心的 OK 按钮上单击。Enterprise Manager 将开始完全数据库文件的恢复过程并显示如图 20.19 所示的 Restore Progress 消息框说明恢复操作的状态。

当 Enterprise Manager 完成从备份文件恢复数据库以后，程序将显示 Restore Completed

Successfully (恢复已经成功完成) 的消息框。看到这个消息框时, 在 OK 按钮上单击。

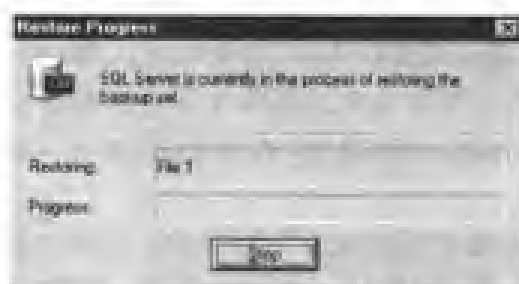


图 20.19 MS-SQL Server Enterprise Manager 的 Restore Progress 消息框

在从完全数据库备份文件恢复数据库以后, 需要应用对储存在随完全数据库备份文件后创建的事务处理日志备份文件中的数据库的改变。要从事务处理日志备份文件恢复并应用改变, 请执行下列步骤:

1. 在含有要恢复其事务处理日志备份文件的数据库的 SQL 服务器图标上单击。例如, 如果要恢复运行于名为 NVBIZNET2 的 NT 服务器上的名为 MSSQL2000 的 SQL 服务上的 COMPANY_DB 数据库的事务处理日志备份文件, 在 Enterprise Manager 左窗格中 NVBIZNET2\MSSQL2000 左侧的图标上单击。Enterprise Manager 将用其右窗格显示包含所选 MS-SQL Server 上可用的数据库和服务的文件夹。
2. 选择 Tools 菜单的 Restore Database 选项。Enterprise Manager 将显示如图 20.14 所示的 Restore Database 对话框。
3. 在 Restore as Database 域右侧的下拉列表按钮上单击并从下拉列表中选择要恢复其事务处理日志的数据库的名称。
4. 在对话框 Restore 部分的 From Device 单选按钮上单击。Enterprise Manager 将在对话框的 Parameters 部分中显示 Restore from Device 选项, 如图 20.15 所示。
5. 在对话框中部的 Device 滚动框右侧的 Select Devices 按钮上单击。Enterprise Manager 将显示如图 20.16 所示的 Choose Restore Devices 对话框。
6. 在 Disk 单选按钮上单击并随后在对话框中部的 Restore from Set 部分的 Add 按钮上单击。Enterprise Manager 将显示如图 20.17 所示的 Choose Restore Destination 对话框。
7. 在 Backup Device 单选按钮上单击。然后在 Backup Device 域右侧的下拉列表按钮上单击, 选择要用来创建事务处理日志备份文件的备份 (转储) 设备。
8. 在靠近 Choose Restore Destination 对话框右下角的 OK 按钮上单击返回 Choose Restore Devices 对话框。
9. 如果要恢复的事务处理日志文件备份储存在多个文件中, 必要时重复第 6 步~第 8 步。
10. 在靠近 Choose Restore Devices 对话框底部右侧的 OK 按钮上单击返回 Restore Database 对话框。
11. 在 Restore Backup Set 单选按钮上单击并随后在对话框下半部的 Transaction Log 单选按钮上单击。
12. 在 Options 上单击显示如图 20.18 所示的域。
13. 查看 Options 选项卡中心的 Restore Database Files As scroll 滚动框的内容验证恢复过程将在其中查找要更新的已有的物理数据库和事务处理日志文件的文件位置的有效性。默认情况

下，Enterprise Manager 将把事务处理日志备份文件恢复到 Enterprise Manager 执行事务处理日志备份时它们所在的文件夹相同的文件夹。如果在恢复完全数据库备份文件时改变了数据库或事务处理日志文件的物理位置，现在就要在 Move to Physical File Name 列中作出同样的改变。

14. 如果有恢复所需要的在恢复当前的事务处理日志备份文件以后的额外的事务处理日志文件，在 Leave Database Nonoperational but Able to Restore Additional Transaction Logs 单选按钮上单击。

15. 在靠近 Restore Database 对话框底部中间的 OK 按钮上单击。Enterprise Manager 将开始事务处理日志备份文件的恢复过程并显示如图 20.19 所示的 Restore Progress 消息框说明恢复操作的状态。

在 Enterprise Manager 已经完成从备份文件恢复事务处理日志以后，它将显示 Restore completed successfully 消息框。当看到这个消息框时，在 OK 按钮上单击。

除了恢复物理的事务处理日志文件外，恢复过程也通过执行储存于事务处理日志文件中的事务处理更新数据库。同样，当完成事务处理日志备份文件的恢复过程时，数据库看起来就像是 MS-SQL Server 执行事务处理日志备份时的样子。

以所需要的频率重复事务处理日志恢复过程以便恢复所有在全数据库备份后随即创建的事务处理日志备份文件。当执行过程中的步骤恢复最终的事务处理日志备份文件时，要确保选择第 14 步中的 Leave Database Operational, No Additional Transaction Logs Can Be Restored 单选按钮。

技巧 406 理解 MS-SQL Server 优化器提示

当向 MS-SQL Server 提供 SQL 语句供执行时，数据库管理系统（DBMS）把语句发送给查询优化器。查询优化器分析语句并生成 DBMS 将执行的查询计划（就是步骤序列）以便执行语句。通过向 SQL 语句添加一个或多个优化器提示，可以改变查询优化器将包括在语句执行计划中的数据提取方法的锁定机制。简而言之，当向语句添加优化器时，就是正在通过确定从数据库提取数据或进行锁定以防止异常的删除或修改的最有效的方式来进行优化器的部分工作。

例如，如果没有优化器提示，执行 SELECT 语句时查询优化器可能决定不使用索引，因为查询只返回少数的行。只从表（相对于从表和索引）提取数据要快些。类似地，如果表不止一个索引，从表中提取数据时需要查询优化器决定使用其中的哪一个，查询优化器将选择它认为将导致最快地提取满足查询条件的数据的索引。在这两种情况下，都可以向查询添加优化器提示以强制查询优化器在其执行计划中使用特定的索引，所以在所选择的索引中返回一组按列的升序储存的行。

也可以用优化器提示控制执行 DELETE、INSERT、SELECT 或 UPDATE 语句时系统的锁定行为。例如，向 SELECT 语句添加 WITH NOLOCK 提示告诉 DBMS 读取当前由其他用户锁定的数据（包括已经插入或更新而没有提交的数据）。类似地，添加 WITH HOLDLOCK 优化器则告诉 DBMS 防止其他用户修改由 SELECT 语句返回的行中的数据，直到当前事务处理完成。

表 20.1 列出了 MS-SQL Server 上可用的优化器提示及每个提示的描述。

表 20.1 MS-SQL Server 优化器提示

优化器提示	描述
FAST x	为快速提取“X”行而优化查询。在返回第一个“X”行以后，查询执行将继续以产生完全的结果集
HOLDLOCK	维持共享锁定直到事务处理完成而不是在读取行、表或数据页中的数据后释放其上面的锁定。使用 HOLDLOCK 与把事务处理的隔离级别设为 SERIALIZABLE 相似
INDEX = x	当选择行时使用 INDEX “x”
INDEX (<index number list>)	按选择行时指定的顺序使用索引
NOLOCK	不设置共享锁定且不执行排他锁定。NOLOCK 只在 SELECT 语句中可用，使查询从没有提交而可能在查询終了前回滚的 UPDATE 或 DELETE 语句读取数据成为可能
PAGLOCK	使用页级的锁定来一次锁住一个表中的一页数据而不是使用行级的或表级的锁定
READCOMMITTED	用 READ COMMITTED 事务处理隔离级别（技巧 269）的语义扫描表中的行
READPAST	跳过被其他事务处理锁定的行而不是等待其他事务处理在完成查询前释放锁定。READPAST 只能应用于 SELECT 语句在 READ COMMITTED 事务处理隔离级别（技巧 268）上的操作，并且只跳过行级的锁定
READUNCOMMITTED	等于执行带 NOLOCK 优化器提示的或在 READ UNCOMMITTED 隔离级别（技巧 269）上的 SELECT 语句
REPEATABLE READ	用 REPEATABLE READ 事务处理隔离级别（技巧 267）的语义扫描表中的行
ROWLOCK	使用行级的锁定来一次锁住一个表中的一行数据而不是使用页级的或表级的锁定
SERIALIZABLE	等于执行在 SERIALIZABLE 隔离级别（技巧 266）上的 SELECT 语句
TABLOCK	设置锁定整个表的表级锁定而不是使用行级或页级锁定
TABLOCKX	设置锁定整个表排他性（相对于默认的共享）表级锁定，而不是使用行级或锁定或页级锁定。执行带 TABLOCKX 优化器提示的语句能防止其他事务处理读取或更新整个表中的数据
UPDLOCK	读取表中的数据时使用更新锁定而不是共享锁定。更新锁定让其他用户通过事务处理读取数据，但防止他们更新数据，直到事务处理終了并释放其更新锁定
XLOCK	设置—排他（相对于共享或更新）锁定。排他锁定防止其他用户访问被其他事务处理读取的数据，直到事务处理終了并释放它所设置的排他锁定

使用 WITH 或 OPTION 子句向 SQL 语句中引入一个或多个优化器提示如下所示：

```
DELETE [FROM] <table name> WITH (<optimizer hints>) ...
```

或

```
DELETE [FROM] <table name>
```

```
 [<WHERE clause>] OPTION (<optimizer hints>)
```

```
INSERT [INTO] <table name> WITH (<optimizer hints>) ...
```

```
SELECT <query expression> [<ORDER BY clause>]
```

```
[<COMPUTE clause>] [<FOR clause>]  
OPTION (<optimizer hints>)  
UPDATE <table name> WITH (<optimizer hints>) ...
```

或者

```
UPDATE <table name> [<FROM clause>]  
OPTION (<optimizer hints>)
```

当选择优化器提示来提高查询效率时，重新查看查询优化器生成的执行计划中的指令清单将对你大有帮助。如果发现了高耗资源的步骤如全表扫描，可以创建新的索引或者利用引导优化器使用已有索引跳过这一步。

如果在开启 MS-SQL Server 的 SHOWPLAN_TEXT 选项（将在技巧 407 中学习）后执行查询，DBMS 将显示查询优化器为 SELECT 语句的执行计划生成的步骤。在查看计划以后，可以试着执行带有不同优化器提示集的相同查询直到查询优化器生成更喜欢的执行计划。

技巧 407 用 MS-SQL Server 的 SHOWPLAN_TEXT 选项显示语句的执行计划

正如在技巧 406 “理解 MS-SQL Server 优化器提示”中提到的，当向 MS-SQL Server 提交 SQL 语句供执行时，DBMS 把语句发送给查询优化器，查询优化器逐个生成语句的执行计划（执行计划是查询优化器告诉 DBMS 为执行语句而要执行的步骤的序列）。

当告诉 MS-SQL Server 执行语句时，通常只对语句的结果感兴趣而对 DBMS 如何产生它们不感兴趣。例如，向 DBMS 提交如下查询：

```
SELECT a.au_lname, a.au_fname, t.title, t.ytd_sales  
FROM authors a, titles t, titleauthor ta  
WHERE a.au_id = t.au_ID AND t.title_ID = ta.title_ID
```

想要 SQL 服务器返回作者姓名、书标题和销售数量的清单。然而，不必真正关心 DBMS 如何从表中提取信息。同样，DBMS 也从视图中隐藏了执行计划。

然而，如果正在试图改进语句的性能，就将要查看 DBMS 为执行它而必须执行的步骤。例如，当优化 SELECT 语句时，就明确地想要知道 DBMS 执行查询时是否必须执行全表扫描。因为读取大表中的每一行通常会使执行查询耗费不可接受的长时间，DBMS 只有如果不能使用已有的索引按所要求的方式提取数据时才会执行全表扫描。所以，查看查询执行计划中的全表扫描将告诉你它有可能创建将减少查询执行时间的索引。

可以用 MS-SQL Server 的 SHOWPLAN_TEXT 选项（以及将在技巧 408 中讨论的 SHOWPLAN_ALL 选项）告诉 DBMS 展示当执行语句批处理中的 SQL 语句时“在后台”会发生什么。要理解的重要事情是在通过向 MS-SQL Server 提交 SET 语句：

```
SET SHOWPLAN_TEXT ON
```

供执行而打开 SHOWPLAN_TEXT 选项后，DBMS 将显示提交给 DBMS 的每个语句的执行计划。然而，DBMS 直到通过执行 SET 语句

```
SET SHOWPLAN_TEXT OFF
```

关闭 SHOWPLAN_TEXT 选项时才实际执行所提交的任何语句。

例如，如果执行 SET 语句：

```
SET SHOWPLAN_TEXT ON
```

并随后提交显示于图 20.20 上部窗格中的语句批处理，DBMS 将返回带的语句批处理中语句的文本结果集，然后把结果集显示在图 20.20 的结果窗格（下部）中。

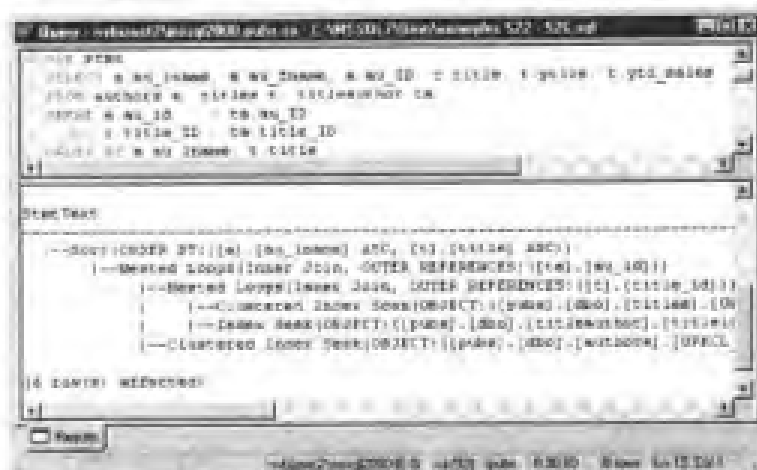


图 20.20 提交带 SHOWPLAN_TEXT 选项的查询后 MS-SQL Server SQL Query Analyzer 的结果

当查看完试图优化的 SQL 语句的执行计划后，执行 SET 语句：

```
SET SHOWPLAN_TEXT OFF
```

告诉 DBMS 停止显示执行计划并恢复提交给服务器的语句的正常执行。

技巧 408 理解显示语句执行计划和状态的 MS-SQL Server SHOWPLAN_ALL 选项

为构建语句的执行计划，MS-SQL Server 查询优化器使用服务器储存的关于数据库表和索引的统计信息来评价执行语句的可供选择的方法。例如，当提交带 ORDER BY 子句的 SELECT 语句时，查询优化器将对其进行检查数据库是否有它可以用于满足子句的排序要求的索引，或者它是否必须添加对它从查询中包含的表中提取的行进行排序的物理步骤。简而言之，查询优化器试图为提交给 SQL Server 供执行的 SQL 语句创建“花费最少”（也就是最有效）的执行计划。

在技巧 407 “用 MS-SQL Server 的 SHOWPLAN_TEXT 选项显示语句的执行计划”中，读者学习了如何把 SHOWPLAN_TEXT 选项设置为“启用”以便 DBMS 显示告诉它要执行的语句的执行计划。把 MS-SQL Server 的 SHOWPLAN_ALL 设置为“启用”也告诉 DBMS 显示语句的执行计划而不是实际执行语句。然而，除了显示执行计划中的步骤外，SHOWPLAN_ALL 选项还为每个步骤提供表 20.2 所示的数据。

表 20.2 SHOWPLAN_ALL 执行计划结果集的每行中的列

列名	描述
StmtText	对类型不为 PLAN_ROW 的行，含 Transact-SQL 语句的文本。对类型为 PLAN_ROW 的行，包含对包括物理操作符（如 Index Seek 或 Clustered Index Scan）和可能的逻辑操作符（如 Compute Scalar 或 Stream Aggregate）的描述
StmtID	当前连接中的语句号。例如，如果正在用 SQLQuery Analyzer 执行执行 SQL 语句并且这是提交给服务器的第 15 个语句，StmtID 的值就是 15
NodeID	当前步骤中节点的 ID（单个步骤可能由多个物理数据库操作或节点组成）
ParentID	当前步骤的父步骤的节点的 ID
PhysicalOp	在节点中执行的物理数据库操作
LogicalOp	由节点中执行的物理操作代表的逻辑操作和有关的代数操作

续表

列名	描述
Argument	关于节点上执行的物理数据库操作的附加信息
DefinedValues	包含由在节点中执行的操作引入的值的由逗号分隔的列表。值可以是计算表达式或查询操作符处理查询所需的内部值。一旦被引入，被定义的值就可以在语句内的其他地方引用
EstimateRows	当前操作符将生成的输出的估计行数
EstimateIO	估计的当前操作的输入/输出的消耗
EstimateCPU	估计的当前操作的 CPU 消耗
AvgRowSize	经过当前操作符传递的每一行中数据的字节数的估计值
TotalSubtreeCost	当前操作及其所有子操作的总消耗的估计值
OutputList	包含由当前操作所计划的列的由逗号分隔的列表
Warnings	包含与当前操作有关的警告消息的由逗号分隔的列表。查询优化器每次不得不以没有数据的列为基础作出决定时都会发出警告消息
Type	包含当前执行计划中父节点的 Transact-SQL 语句的类型和节点的 PLAN_ROW
Parallel	如果操作符没有并行运行，则为零（0），否则为一（1）
Estimated Executions	本操作将在当前语句中执行的次数的估计值

与 SHOWPLAN_TEXT 选项的情况一样，当重新查看完 SQL 语句的执行计划并想要 DBMS 停止显示执行计划并恢复所提交的 SQL 语句的正常执行时，必须执行 SET 语句：

```
SET SHOWPLAN_TEXT OFF
```

技巧 409 使用 MS-SQL Server SQL Query Analyzer 的 SHOWPLAN 选项

除了生成带有执行计划信息的基于文本的结果集外，MS-SQL Server SQL Query Analyzer 也允许以图形方式显示执行计划。在技巧 407 和技巧 408 中，已经学过如何使用语句 SET SHOWPLAN_TEXT ON 和 SET SHOWPLAN_ALL ON。使用这些语句时 DBMS 返回由查询优化器生成的 SQL 语句的执行计划而不是执行语句并返回其结果集。

当向 MS-SQL Server 提交带 SHOWPLAN_TEXT ON 或 SHOWPLAN_ALL ON 的 SQL 语句供执行时，DBMS 返回列出了如果它要执行该语句时将执行的步骤的执行计划。DBMS 把执行计划作为表中行的列值返回——就像它将返回由被服务器执行的语句生成的结果集一样。同样，在 SHOWPLAN_TEXT 或 SHOWPLAN_ALL 开启的情况下，可以在 SQL-Query Analyzer 的查询（顶部）窗格中键入 SQL 语句批处理，选择 Query 菜单的 Execute 选项，并在 SQLQuery Analyzer 的结果（底部）窗格中重新查看语句的执行计划。

也可以通过执行下列步骤用 MS-SQL Server 的 Query Analyzer 以图形方式显示语句的执行计划：

- 1. 通过在 Windows 的 Start 按钮上单击启动 SQLQuery Analyzer。当 Windows 显示 Start 菜单时，把鼠标指针移到 Programs，选择 Microsoft SQL Server 2000，然后在 Query Analyzer 上单击。然后，Windows 将启动 SQLQuery Analyzer，后者将显示 Connect to SQL Server 对话框。
- 2. 用对话框顶部 SQL Server 域右侧的下拉列表按钮选择希望登录的 SQL Server。在 Login

name and Password 域中输入用户名和密码。然后,在靠近对话框底部中间的 OK 按钮上单击完成登录过程。

3. 用 SQL Query Analyzer Standard 工具栏上的下拉列表按钮(从右开始的第6个按钮)选择希望操作的数据库。对当前项目而言,就是选择 PUBS。

4. 在 SQL Query Analyzer 应用程序窗口顶部的查询窗格中输入希望查看其执行计划的 SQL 语句。对当前项目而言,就是输入 `SELECT * FROM authors ORDER BY au_lname`。

注意:如果在查询窗格中键入多于一个语句,可以把所有语句作为语句批处理生成执行计划。相反,如果想只为查询窗格中几个语句中的一个生成执行计划,在执行第5步前用鼠标指针选择要查看其执行计划的语句。

5. 要显示 SQL Query Analyzer 查询窗格中语句(或语句批处理)的执行计划而不执行语句,可选择查询菜单的 Display Estimated Execution Plan 选项。SQLQuery Analyzer 将用其结果(在底部)窗格显示查询(顶部)窗格中语句(或语句批处理)的执行计划,如图 20.21 所示。

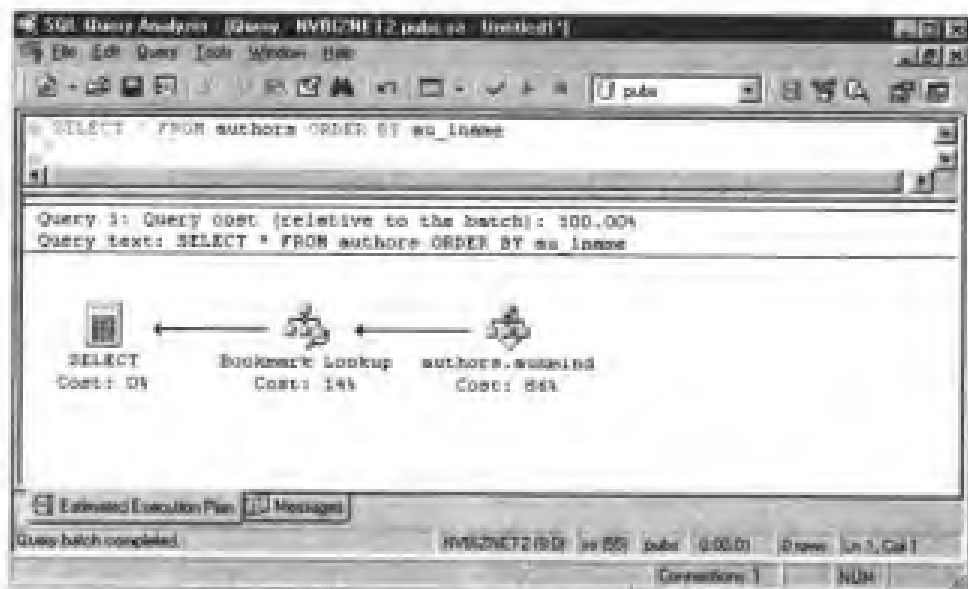


图 20.21 SQL Query Analyzer 的语句执行计划的图形显示

注意:如果想要 SQL Query Analyzer 既向 DBMS 提交查询窗格中的语句(或语句批处理)供执行又显示执行计划,可选择 Query 菜单的 Show Execution Plan 选项。不幸的是,SQLQuery Analyzer 不会打开第二个结果窗格以便查看查询结果和执行计划。所以,如果在查询窗格中输入 SELECT 语句并选择 Query 菜单的 Show Execution Plan 选项,SQL Query Analyzer 将请求 DBMS 执行查询。然而,请记住在屏幕上只能看到一个执行计划(而不查询的结果集)。

显示于结果窗格中的图形化执行计划中的每个图标都代表计划中的一个步骤。当把鼠标指针移到图标上时,SQL Query Analyzer 将显示计划中该步骤的几个 SHOWPLAN_ALL 列(在表 20.2 中解释过)的值,如图 20.22 所示。类似地,如果把鼠标指针移到图标之间的箭头上,SQLQuery Analyzer 将显示从一个步骤传递到下一步骤的估计数或行数以及每行传递的平均字节长度的估计值。

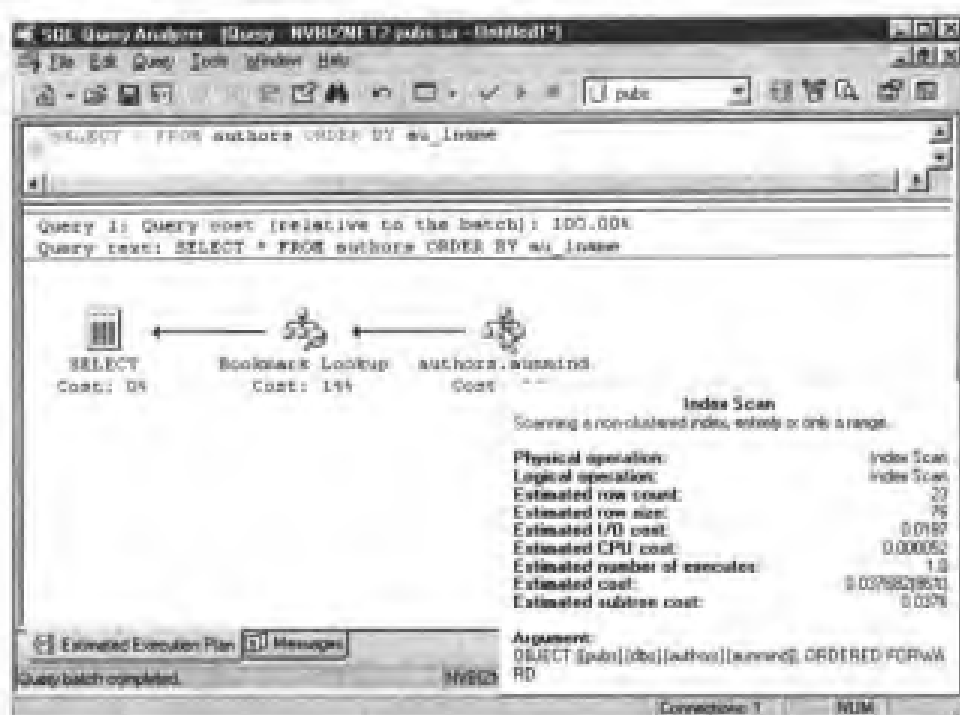


图 20.22 SQL Query Analyzer 图形化执行计划的详细资料

技巧 410 用 MS-SQL Server SETUSER 语句测试用户对数据库对象的访问权限

当实现数据库安全时，将给予用户以能访问某些数据库而不能访问其他数据库的访问权限。要测试用户的访问权限，可登录到用户账号，或者如果是系统管理员（sa）或数据库所有者（DBO），可以使用 SETUSER 语句来扮演任何的数据库用户。

SETUSER STATEMENT 语句的句法如下：

SETUSER <username> [WITH NORESET]

所以，例如，当登录到 sa 账号时就可以执行如下语句：

SETUSER MARY

测试 MARY 对各个数据库对象的访问权限。而且，用户名 MARY 将拥有当 DBMS“认为”你是 MARY 时创建的任何数据库对象。同样，SETUSER 给出了创建新的数据库对象并给予特殊用户以对这些对象的所有访问权限（作为数据库对象的所有者[DBO]）。

当要回复成为 sa 或 DBO 时，可执行不带用户名的 SETUSER 语句，或执行 USE 语句。

如果在 SETUSER 语句中包括了 WITH NORESET 子句，如

SETUSER MARY NORESET

来扮演另一个用户，DBMS 将不把你的身份改回到原先登录时所用的账号（sa 或 DBO）。如果执行 SETUSER 语句时包括了 NO RESET，回到原先身份的惟一方法是退出并作为 sa 或 DBO 重新登录。

第 21 章 使用存储过程

技巧 411 理解存储过程

正如本书从头到尾经常提到的，标准 SQL 并不是完全的编程语言，SQL 是非过程的、数据子语言，其中的语句允许用户创建并删除数据库和数据库对象；指定访问权限；输入、更新并删除数据以及通过查询提取要向外应用程序输出的数据。由于 SQL 语句常常嵌入程序或使用应用程序编程接口（API）作为函数调用来执行，所以有时还难以区分何处是 SQL 以及何处是编程语言的过程部分。

但是，标准 SQL 是面向数据的而且没有控制循环的结构、没有条件执行的关键词，甚至没有作为语句块执行多条语句的机制。确实如此，可以作为打开的事务处理的一部分来执行多条语句。虽然编程语言允许指定一套语句按指定的多步任务的顺序来执行，但 SQL 的事务处理只识别由一套既可取消也可向数据库永久提交的自主语句所完成的工作。

一个存储过程，与事务处理不同，是以特定的顺序排列的 Transact-SQL 语句序列，可为其指定名称、加以编译并保存在 MS-SQL Server 上。一旦由 DBMS 编译并保存，用户可使用应用程序（如 MS-SQL Server 的 SQL Query Analyzer）通过单条命令告诉 MS-SQL Server 执行存储过程中的语句——这类似于在应用程序中调用子程序。另外一点与应用程序中的子程序类似的地方是，存储过程给予人们一种以特定顺序完成多步任务而执行一套 SQL 语句的方法。简而言之，存储过程允许使用 SQL 语句来编写“程序”——或者至少是在数据库本身之内可调用来完成与数据库有关的应用程序处理过程。

例如，可以使用存储过程把现金从信用卡或储蓄账户上转移到支票账户，这就可避免当支票开出用于支付时，顾客的支票账户的余额有可能成为负数。类似地，存储过程可以被调用以便接受顾客定单，这就要求更新存货目录表中几种产品的存货数量，然后向发货和发票表中插入记录，当然也要更新顾客和销售人员表上的余额。

这样一来，存储过程为标准 SQL 增加了几种功能，这些功能通常都与编程语言有关（而这正是 SQL 数据子语言所缺少的）。这些添加的功能包括：

- 条件执行——在存储过程中放入一套 Transact-SQL 语句之后，可使用 Transact-SQL 的 If-Then-Else 结构根据存储过程中其他语句的执行返回的结果来决定执行过程中的哪条语句。
- 循环控制结构——Transact-SQL 的 WHILE 和 FOR 语句允许重复地执行一系列语句，直到满足某种终止条件为止。
- 命名变量——可以在存储过程中使用命名的内存位置（也就是变量）来保存通过参数传向过程的值，以及由过程内的查询返回的值或是由某些其他方法计算的值。
- 命名过程——在存储过程中放入一条或多条 Transact-SQL 语句并添加了所希望的 Transact-SQL 的条件执行和循环控制结构之后，可为存储过程起一个名称并通过正式的输入和输出参数把数据传入过程或从过程传出来。简而言之，存储过程有与过程的

或面向对象的编程语言中的子程序一样的外观与感受。另外，一旦定义和编译之后，可通过名称在触发器中调用，在交互环境中通过 MS-SQL Server SQL Query Analyzer 调用，也可以在应用程序中调用，甚至还可作为标准 SQL 语句的子句或判式来使用。

- 语句块——正如在应用程序中调用子程序一样，通过调用存储过程可让 DBMS 执行一系列 SQL 语句，就像执行单条语句一样，但却可完成几种不同而有联系的任务。

除了扩展了标准 SQL 的功能使之比单纯的数据子语言更像是编程语言之外，存储过程还提供几种比交互式的 SQL 语句执行更为优越的地方：

- 封装——在面向对象的编程领域里，存储过程是可用于操作数据库对象的方法（methods）。使用 SQL 的安全性不允许对 SQL 语句和数据库对象的直接访问，从而在存储过程中既可封装语句也可封装对象。强制每个人只能使用存储过程来处理数据库内的数据可防止用户通过在使用数据库的应用程序中不应用规则或是跳过完整性检查来绕过商业规则。另外，存储过程使得用户在不了解数据库对象甚至不了解 SQL 语句执行的情况下安全地使用数据库成为可能。为了调用存储过程，用户只需要了解过程调用中的输入和输出参数并理解存储过程的目的。
- 改善性能——每当向 MS-SQL Server 提交供执行的 SQL 语句时，DBMS 在可执行语句之前，必须分析、确认、优化并事先为存储过程中的整个语句序列生成执行计划。由此一来，当在运行时调用存储过程时，DBMS 可快速执行存储过程中的语句，因为可直接转到语句的执行，无需分析、确认、优化和生成执行计划的步骤，这些步骤先前就已经完成了。
- 减少了网络流量——每当向 DBMS 提交供执行的 SQL 语句时，工作站必须通过网络向 MS-SQL Server 发送语句，而 DBMS 必须向工作站返回语句的结果集。如果有生成大量结果集的语句批处理存在，在工作站和 SQL 服务器间的这一来一回的网络流量可导致网络拥挤。作为对照，如果调用包括相同语句批处理的存储过程，DBMS 在服务器上执行语句并只向调用存储过程的工作站发送回最后的结果集，或许只有单一值。
- 重用性——在编译了存储过程之后，许多用户和应用程序都可执行相同的语句序列，而不必重新键入和重新向 DBMS 提交。当执行需要许多 SQL 语句或是具有复杂逻辑的语句时，执行已经调试并测试过的 SQL 语句批处理减少了引入程序员错误的风险。
- 安全性——如果数据库所有者（DBO）或系统管理员（sa）编译并保存了存储过程，存储过程就有了对它所使用的数据库对象的所有的访问权限。因而，系统管理员可向单独的用户授予对数据库对象的最小访问权限。不是直接允许用户使用数据库对象，SA 通过授予用户对存储过程的执行权限，就可控制工作完成的方式。存储过程使用户只能以事先批准的方式（即在存储过程中定义的方法）来操作数据。

技巧 412 使用 CREATE PROCEDURE 语句创建存储过程

CREATE PROCEDURE 语句允许创建、编译并在 MS-SQL Server 上保存存储过程。在默认情况下，只有数据库所有者（DBO）具有对数据库的 CREATE PROCEDURE 访问权。但是，DBO 可执行以下形式的 GRANT 语句向由语句中的 <username> 所标识的用户 ID 授予 CREATE PROCEDURE 访问权：


```
GRANT CREATE PROCEDURE <username>
```

CREATE PROCEDURE 语句的句法如下:

```
CREATE PROCEDURE <procedure name> [; <version number>]  
[[@<parameter name> <data type> [VARYING]  
[=<default value>] [OUTPUT]  
[... ,n]]]  
[WITH {RECOMPILE|ENCRYPTION|RECOMPILE, ENCRYPTION}]  
[FOR REPLICATION]  
AS <Transact-SQL statement(s)>
```

其中:

- <procedure name>是存储过程的名称, 名称至多可有 128 个字符。
- <version number>允许创建多个有相同名称的存储过程。通过在执行命令中包括版本号从而执行指定版本的存储过程, 如下面的语句所示: EXEC usp_proc;2。如果调用存储过程时没有指定版本号, DBMS 将执行具有相同名称的存储过程组中的最高版本号的存储过程。通过在 DROP 语句中包括版本号, 如 DROP PROCEDURE usp_proc;2 所示, 可以删除特定版本号的存储过程, 或者忽略版本号, 如 DROP PROCEDURE usp_proc 从而一次删除存储过程的所有版本。
- <parameter name>是可用作存储过程中的变量的参数名。每个参数的值必须在存储过程调用中提供或者作为 CREATE PROCEDURE 语句的一部分设置为缺省值。虽然参数可用作存储过程中的变量名, 但参数不能用作列名、表名或是其他数据库对象的名称。
- <data type>是参数的数据类型。参数可以是任何合法的 SQL 数据类型 (包括 TEXT、NTEXT 和 IMAGE) 或用户定义的数据类型。如果参数是 CURSOR 数据类型, 该参数还必须被指定为 VARYING 和 OUTPUT。
- VARYING 仅对数据类型为 CURSOR 的参数才是合法的。指明该参数将包括内容会变化的结果集, 其内容由存储过程中的语句动态地构成。
- <default value>是参数的默认值。如果指定, 则过程可在不指定参数的情况下执行。
- OUTPUT 指明参数在存储过程中可以变化, 而且修改后的值可返回给主调过程。
- ...,n 指明 CREATE PROCEDURE 语句可有多达 2100 个参数。
- RECOMPILE 每当存储过程被调用时告诉 MS-SQL Server 都要进行编译 (也就是生成新的执行计划)。如果没有此选项, DBMS 在执行 CREATE PROCEDURE 语句时编译存储过程, 每次调用时都使用同一执行计划。
- ENCRYPTION 告诉 MS-SQL Server 加密 SYS COMMENTS 表中存储过程条目的文本, 以防止用户查看编译后的存储过程中的语句。指定此选项还可防止存储过程被作为 MS-SQL Server 的复制过程而分开。
- FOR REPLICATION 是只能在复制过程中执行的存储过程。不能在同一 CREATE PROCEDURE 语句中同时指定 RECOMPILE 和 FOR REPLICATION 两者。
- <Transact-SQL statements>是 Transact-SQL 语句。存储过程的最大容量为 128MB。

例如, 为了创建显示 EMPLOYEES 文件内容的存储过程, 可执行如下的 CREATE PROCEDURE 语句:

```
CREATE PROCEDURE usp_show_all_employees
```

```
AS SELECT * FROM employees
```

注意：应该遵循标准的存储过程的命名约定。除非在 MASTER 数据库中与其他存储过程一起创建并保存存储过程，在命名自己创建的存储过程时，应当避免使用系统存储过程使用的“sp_”前缀。在接收到执行以“sp_”开头的存储过程的命令之后，MS-SQL Server 首先在 MASTER 数据库内寻找存储过程。这种搜索是相当耗时的（如果 MASTER 数据库中有许多存储过程的话），即使不是如此也是无用的，因为想要执行的存储过程并不位于 MASTER 数据库中。请使用用户存储过程的标准前缀“usp_”。如果这样做了，当用户调用一个存储过程时，DBMS 搜索当前数据库，这就可避免首先搜索 MASTER 数据库的时间浪费。

当执行 CREATE PROCEDURE 语句时，DBMS 分析、确认、优化并生成存储过程中每个语句的执行计划。如果任何存储过程中的语句在句法不正确或是引用了不存在的数据库对象（不是存储过程），DBMS 将拒绝该存储过程并报告所发现的错误，从而用户可改正语句并重新向 DBMS 提交改正后的 CREATE PROCEDURE 语句供执行。如果存储过程中的一个或多个语句引用了其他的但还没有实现的（即当前还不存在的）存储过程，DBMS 将生成警告消息，但会仍然编译并在 MS-SQL Server 上安装该存储过程。

注意：当确认存储过程中的语句时，DBMS 并不考虑对存储过程的语句中所引用的数据库对象的访问权限。因而，用户可能会执行一条 CREATE PROCEDURE 语句创建出该用户其后不能执行的存储过程。

技巧 413 用 EXECUTE 语句调用存储过程

在创建一个存储过程之后（正如上一个技巧中所学过的），可通过以下方式告诉 DBMS 执行存储过程中的 Transact-SQL 语句：

- 在应用程序中，提交调用存储过程的 SQL 语句执行请求。
- 作为语句批处理的第一行，输入存储过程名。
- 在交互会话期间，在另一个存储过程或者触发器中，提交指定存储过程的 EXECUTE（或 EXEC）语句。
- 在 SQL 语句中，把存储过程的名字作为参量。

例如，调用一个没有参数的存储过程（如在前一技巧快要结束时创建的 USP_SHOW_ALL_EMPLOYEES 存储过程），只要输入 EXECUTE（或 EXEC），后面再键入一个空格和例程名，如下所示：

```
EXECUTE usp_show_all_employees
```

或者：

```
EXEC usp_show_all_employees
```

在向 DBMS 提交前面两个语句中的任何一个语句供执行之后，DBMS 将执行名为 USP_SHOW_ALL_EMPLOYEES 的存储过程中的语句。

如果自己不拥有存储过程，但只要被授予对该例程的 EXECUTE 权限，则必须在 EXECUTE 语句中输入拥有者的用户名，如下所示：

```
EXEC frank.usp_show_all_employees
```

这就告诉 DBMS 执行由用户名 FRANK 所拥有的 USP_SHOW_ALL_EMPLOYEES 存储过程。

注意：如果有对数据库所有者（DBO）的存储过程的 EXECUTE 权限，在使用 EXECUTE

语句调用该例程时,在存储过程名称中不必指定 DBO,DBMS 自动地搜索自己所拥有以及 DBO 所拥有的存储过程列表。

如果存储过程有参数清单,可在 EXECUTE 语句中的过程名之后包括参数值。参数值是按存储过程的 CREATE 语句中参数出现的先后顺序列出的。请用逗号分隔参数并将字符串用引号括起来。

例如,为了调用由下面语句所建立的存储过程:

```
CREATE PROCEDURE usp_add_employee @first_name VARCHAR(30),
    @last_name VARCHAR(30),
    @address VARCHAR(30),
    @office INTEGER = 1,
    @manager INTEGER = 1
AS INSERT INTO employees
    (first_name, last_name, address, office, manager)
VALUES
    (@first_name, @last_name, @address, @office, @manager)
```

可提交如下 EXEC 语句:

```
EXEC usp_add_employee 'Wally', 'Wallberg',
    '777 Sunset Strip', 5
```

如果愿意的话,可在过程调用中明确指定参数名,这样就可以想要的任何顺序列出参数。例如,以下过程调用与前面的由位置指定参数值的过程调用是等价的:

```
EXEC usp_add_employee @office = 5,
    @address = '777 Sunset Strip',
    @first_name = 'Wally', @last_name = 'Wallberg'
```

技巧 414 使用存储过程参数返回值

在以上 3 个技巧中,读者已经学习了如何创建并调用存储过程。正如技巧 411 “理解存储过程”中所学习的,可使用存储过程按照特定的顺序安排来执行语句序列。一旦编译并由 DBMS 加以储存,则可使用关键词 EXEC (或者 EXECUTE) 后跟存储过程名向 DBMS 提交此语句,从而执行存储过程中的语句。

例如,为了执行存储过程 USP_PROCESS_CHECK,可向 DBMS 提交以下语句:

```
EXEC usp_process_check 112233, 123, 258.59
```

可通过参数向存储过程中的语句传递值。存储过程的参数同其他变量一样,是可保存值的命名的内存位置。在本例中,112233、123 和 258.59 是输入参数,因为它们传入存储过程的值。由此,如果给定以下声明,则本例中的存储过程调用把 112233 作为 @ACCOUNT_NUMBER、123 作为 @CHECK_NUMBER,而 258.9 作为 @CHECK_AMOUNT 传递给存储过程中的语句:

```
CREATE PROCEDURE usp_process_check @account_number INTEGER,
    @check_number INTEGER,
    @check_amount REAL
AS
INSERT INTO checks (@account_number, @check_number,
    @check_amount)
```

虽然输入参数允许将值传入存储过程，但可用输出参数把值返回给主调程序。假设本例中的存储过程，除了把支票信息储存在 CHECK 表中之外，还更新了顾客的支票和贷款余额。那么存储过程的声明可如下所示：

```
CREATE PROCEDURE usp_process_check
  @account_number INTEGER,
  @check_number INTEGER,
  @check_amount REAL,
  @checking_balance REAL OUTPUT,
  @loc_used REAL OUTPUT,
  @loc_balance REAL OUTPUT
AS
SET @loc_used = 0
/* Retrieve current account balances */
SELECT @checking_balance =
  (SELECT checking_balance FROM customers
   WHERE account_number = @account_number)
SELECT @loc_balance =
  (SELECT loc_balance FROM customers
   WHERE account_number = @account_number)
/* If check amount would overdraw the checking balance,
tap the line of credit */
IF @checking_balance < @check_amount
SET @loc_used = (@check_amount - @checking_balance)
/* Store the check within the CHECKS table and update the
customer's balance(s) */
SET @loc_balance = @loc_balance + @loc_used
SET @checking_balance =
  @checking_balance - @check_amount + @loc_used
INSERT INTO checks
VALUES (@account_number, @check_number, @check_amount)
UPDATE customers
SET loc_balance = @loc_balance,
   checking_balance = @checking_balance
WHERE account_number = @account_number
```

在本例中，@CHECKING_BALANCE、@LOC_USED 和 @LOC_BALANCE 都是输出参数，这就意味着，存储过程将把执行它对这些参数值所产生的变化传递给激活存储过程的语句。不要过分地关注本例中存储过程所执行的具体任务。要理解的重要事情是，Transact-SQL 允许指定输入和输出参数。仅当在 CREATE PROCEDURE 语句中，关键词 OUTPUT 跟在参数的数据类型声明之后时，才表明这个参数是输出参数（也就是说，能够将更新了的值传回存储过程的主调程序）。

似乎每个 DMBS 的产品都有自己指定输出参数的方式。例如，Oracle 在参数类型说明之前把参数标示为 in、out 或 in out。Informix 用关键词 RETURNING 指定要返回值的参数。这样一来，如果正在使用的 DMBS 不是 MS_SQL Server，请一定要检查系统文档，找出在存储过程的声明中指定输入和输出参数的特定方法。

注意：可通过输入和输出参数把值传递到存储过程中。但是，如果存储过程改变了输入参数的值，当存储过程执行完毕后，这些参数的新值不会返回到主调程序。相反，存储过程对输出参数的任何改变都可传递回来。因此，输入参数在效果上只是一种单向阀门，允许把值传递给存储过程，但不允许从该过程中提取值。作为对照，输出参数却是一种双向阀——初始值传入存储过程，而更新后的值返回主调程序。

为了允许存储过程中的语句更新输出参数的值，过程调用必须有一个可接受的“目标”，通过该目标能接收到返回的值。

例如，可用 EXEC 语句调用 USP_PROCESS_CHECK，如下所示：

```
EXEC usp_process_check 112233, 123, 250.59,  
    @checking_balance OUTPUT, @loc_balance OUTPUT,  
    @loc_used OUTPUT
```

为了传递输入参数的值，既可用实际值（如在本例中传递的前 3 个参数），也可以传递变量名称，其中包括在输入参数中想要传递的值。然而，对于输出参数，必须使用变量名（也就是命名的内存位置而不是实际值）。因为在将控制权返回存储过程的主调程序之前，存储过程必须具有对放置有参数值的内存位置有访问权。此外，Transact-SQL 要求，在调用存储过程时，必须在允许存储过程改变的每个变量后面包括关键词 OUTPUT。如果省略了关键词 OUTPUT，DBMS 仍将执行程序调用而没有差错。然而，输出参数仍保留着在存储过程执行之前的值，甚至在存储过程内改变了该参数之后仍然如此。

正如以前指出的那样，虽然某些参数指定为 Transact-SQL 的“输出”参数，但输出参数的通讯却不是真正意义的单向方式。事实上，Transact-SQL 把输出参数的当前值传递给存储过程，也把更新后的值传递回存储过程的主调程序。因此，在本例中，如果从存储过程中删除前两条 SELECT 语句，而在调用存储过程之前执行这两条语句，可把@CHECKING_BALANCE 和@LOC_BALANCE 中的余额传递给存储过程。不论初始设置是在存储过程之内还是在存储过程之外，当存储过程将控制权返回给主调程序时，这两个输出参数将保存着更新后的余额信息（分别对于支票账户和信用贷款的最高限额）。

技巧 415 用关键词 RETURN 从存储函数中返回一个值

当想要执行一个语句序列来产生标量（单一）值或表时，可使用存储函数而不是存储过程。与存储过程不同，存储过程可通过输出参数返回多个值，而存储函数只能返回单一值或表（表里可能有多个值）。此外，函数不能执行 INSERT、UPDATE 和 DELETE 或其他修改存储函数外部的数据库对象。然而，如果存储函数返回一个表，那么它能执行改变函数返回的临时表的 INSERT、UPDATE 或 DELETE 语句。

这样一来，如果想改变永久性数据库对象的内容或者结构，或者通过参数返回多个值，则应使用存储过程。如果想返回单个值或在一个表中返回一组值，而不想改变任何永久性的数据库对象，则应使用函数。

为了创建一个函数，可以使用 CREATE FUNCTION 语句，这种语句的句法如下：

```
CREATE FUNCTION [owner_name.]<function name>  
    ([{@<parameter name> [AS]  
        <data type>[=<default values>]}[...n]})  
RETURNS <scalar data type>
```

```

[AS]
BEGIN
    <Transact-SQL statement(s)>
    RETURN <scalar expression>
END

```

在技巧 417 “使用 CREATE FUNCTION 语句创建存储函数”中，读者将学习 3 种形式的 CREATE FUNCTION 语句的每一种的句法。句法的不同取决于函数返回值的类型的不同——一种句法用于返回标量值（如本技巧的以下代码所示），第二种句法用于返回一个内联表，第 3 种句法用于返回一个多语句表）。

目前，只要注意到跟在存储函数参数清单后面的关键词 RETURNS。当存储函数返回单一值（标量）时，可在关键词 RETURNS 之后指定返回值的数据类型。也要注意，RETURN 语句紧在关键词 END 之前，END 结束函数的声明。对于返回标量值的函数，可紧跟在关键词 RETURN 之后，既可作为实际值也可以作为一个包含值的变量或者作为一个表达式指定函数的返回值。

例如，假定想获得那些在被雇佣期间销售量超过 10 万美元的售货员的清单。可提交一个带有相关子查询的 SELECT 语句（正如读者在技巧 253“理解用比较运算符引入的相关子查询”中所学过的）。或者在一个函数中把子查询封装为一个简单的 SELECT 语句。例如将返回一个雇员的总的销售量的函数命名为 UFN_GET_TOTAL_SALES，则查询将同以下所示的类似：

```

SELECT first_name, last_name,
       dbo.ufn_get_total_sales(salesrep_ID)
FROM employees
WHERE dbo.ufn_get_total_sales(salesrep_ID) > 100000.0

```

给出以下定义，每当调用时，存储函数 UFN_GET_TOTAL_SALES 返回单一值（特定雇员的总的销售量）：

```

CREATE FUNCTION ufn_get_total_sales(@salesrep_ID INTEGER)
RETURNS REAL
BEGIN
    RETURN (SELECT SUM(order_total)
            FROM cust_orders
            WHERE cust_orders.salesrep_ID = @salesrep_ID)
END

```

这样，仅当某一销售代表的总订单（存储函数返回）大于 10 万美元时，原来查询中的 WHERE 子句才满足条件。

如本技巧前面所述，可编写返回数据值表而不是单一（标量）值的函数。例如，假定想要获得特定顾客的所有支票和存款清单。可创建如下函数：

```

CREATE FUNCTION ufn_cust_trans(@cust_ID INTEGER)
RETURNS @trans_list TABLE
    (trans_date DATETIME,
     trans_ID INTEGER,
     trans_type CHAR(1),
     trans_amount REAL)
BEGIN
    INSERT @trans_list

```

```
SELECT deposit_date, deposit_ID, 'D', deposit_amount
FROM deposits
WHERE deposits.cust_ID = @cust_ID
ORDER BY deposit_date

INSERT @trans_list
SELECT check_date, check_number, 'C', check_amount
FROM checks
WHERE checks.cust_ID = @cust_ID
ORDER BY check_date

RETURN
END
```

在调用之后，UFN_CUST_TRANS 将按日期顺序返回一个所有存款的表，其后将按支票日期返回所有支票的表。请注意，当用于返回一个表时，存储函数就像是数据库的视图。然而，如本例所示，虽然数据库视图必须基于单条 SELECT 语句，但可使用多个 SELECT 语句（以及执行其他的 Transact-SQL 处理）以生成由存储函数返回的表。

可以像对任何其他“普通”数据库表一样，对函数所返回的表进行查询。例如，可执行以下查询列出表内的所有的事务处理：

```
SELECT * FROM ufn_cust_trans(123456)
或者，可以使用总计函数来显示总存款、支票以及各项的总计额：
SELECT trans_type, COUNT(*), SUM(trans_amount)
FROM ufn_cust_trans(123456)
GROUP BY trans_type
```

注意：在调用返回标量值的用户自定义函数时，必须包括拥有者的姓名——即使存储函数是在自己用户名下由自己创建的。同存储过程不同，如果在函数调用时省略拥有者的 ID，那么 DBMS 既不查找自己拥有的存储函数列表，也不查找 DBO 所拥有的存储函数列表，以便匹配名称。因而如果像下面这样在函数调用中省略了拥有者 ID：

```
PRINT ufn_get_total_sales(1)
DBMS 将报告以下错误：
'ufn_get_total_sales' is not a recognized function name'
```

但是，如本技巧前面第二个例子所示，如果函数返回一个表（而不是标量值），就可以调用自己的或者 DBO 所拥有的函数，而必不提供所有者 ID。

技巧 416 在存储过程中使用游标

在关系型数据库内，通常一次处理一套完整的行集。例如，当执行一条 SELECT 语句时，DBMS 同时返回所有满足查询的 WHERE 子句中的搜索条件的所有行，而不是一次返回一行。类似地，一条 UPDATE 语句对满足其 WHERE 子句中的搜索条件的所有行作出相同的变化。然而，有时，必须一次一行（或者几行）地处理数据，而不是同时处理结果集中的所有行。当必须一行一行地处理结果集时，可以在存储过程内使用数据库游标。

例如，假定有一个存储过程 USP_CLEAR_CHECKS，要用这个例程清除所有顾客的支票。为了减少透支的数目，想让该例程按支票数额升序排列支票（这样，如果顾客的余额不能满足

所有签发的支票的总额时，最大数额的支票就被清除）。在本例中，想让支票最终在以下两个表之一中：CLEARED_ITEMS 或 OVERDRAFT_ITEMS，而且想要在 FEE 表中对每笔透支插入 10 美元的费用。

为了在存储过程（或在一个存储函数）中创建并使用数据库游标，可使用在技巧 331~技巧 346 中学习过的同样的 DECLARE CURSOR、OPEN、FETCH 和 CLOSE 等语句。例如，可声明 USP_CLEAR_CHECKS 过程如下：

```
CREATE PROCEDURE usp_clear_checks
@account_no INTEGER,
@date_processed DATETIME,
@checking_balance MONEY OUTPUT
AS
/* Create temporary variables into which to FETCH values
with the columns values from a table row */

DECLARE @check_amt MONEY
DECLARE @check_date DATETIME
DECLARE @check_no INTEGER

/* Declare the cursor in which to store the query results
temporarily during row-by-row processing */

DECLARE cur_unproc_checks CURSOR FOR
SELECT check_date, check_no, check_amount
FROM unprocessed_checks
WHERE account_no = @account_no
ORDER BY check_amount

/* OPEN the cursor and then FETCH the first row within the
results set */
OPEN cur_unproc_checks
FETCH cur_unproc_checks
INTO @check_date, @check_no, @check_amt

/* If no rows within the results set (because the customer
has no outstanding checks) CLOSE the cursor and RETURN
to the caller */

IF (@@fetch_status <> 0)
BEGIN
CLOSE cur_unproc_checks
DEALLOCATE cur_unproc_checks
RETURN
END
SET NOCOUNT ON
```



```

/* Work through the cursor one row (check) at a time. Check
to make sure the last FETCH was successful, then process
the row of results. After processing, FETCH the next row
and repeat the process until the FETCH is unsuccessful meaning
there are no more rows to process. */

WHILE (@@fetch_status = 0)
BEGIN
IF @checking_balance - @check_amt >= 0
BEGIN
SET @checking_balance =
@checking_balance - @check_amt

INSERT INTO cleared_items
VALUES (@date_processed, @account_no,
@check_date, @check_no, @check_amt)
END
ELSE
BEGIN
INSERT INTO overdraft_items
VALUES (@date_processed, @account_no,
@check_date, @check_no, @check_amt)

INSERT INTO fees
VALUES ('OD', @date_processed, @account_no,
@check_date, @check_no, @check_amt,
10.00)
END

/* FETCH the next row from the results set */

FETCH cur_unproc_checks
INTO @check_date, @check_no, @check_amt
END

CLOSE cur_unproc_checks
DEALLOCATE cur_unproc_checks
RETURN

```

上面的代码代表在 MS-SQL Server 上可用于处理游标结果的 Transact-SQL 语句。如果使用的是不同于 MS-SQL Server 的 DBMS 产品，语句可能要发生变化。例如，在 Oracle 中，应该使用 FOR 循环一次一行地在游标中移动，并直接地处理游标列值，而不必首先把这些值送到中间变量中。另一方面，Informix 将使用 FOREACH 循环把游标内的值行一次一行地转移到临时变量中进行处理。因而，请检查一下具体的 DBMS 系统手册，从而找出在处理游标时必须使用的准确的语句。虽然准确的语句可能不同，但一般来说，游标处理要按以下步骤：

1. 使用 DECLARE CURSOR 语句，把游标与一条 SQL SELECT 语句联系起来（游标实际

上是虚拟表，允许人们一次一行地处理查询返回的结果集）。

2. 向 DBMS 提交一条 OPEN(cursor)语句，以便执行游标的 SELECT 语句并用 SELECT 的结果集来填充游标。

3. 执行一系列的 FETCH 语句，提取游标内的行并在每次提取之后加以处理。

4. 使用 CLOSE(cursor)语句以“关闭”游标，然后 DEALLOCATE 游标以便将其删除，从而释放用于暂时保留 SELECT 语句的结果集的存储空间（也许是硬盘资源）。

技巧 417 使用 CREATE FUNCTION 语句创建存储函数

在 MS-SQL Server 上，使用 CREATE FUNCTION 语句允许创建、编译和保存存储过程。在默认情况下，只有数据库所有者（DBO）有对数据库的 CREATE FUNCTION 访问权。然而，DBO 可执行以下形式的 GRANT 语句：

```
GRANT CREATE FUNCTION <username>
```

向由<username>标识的用户 ID 授予 CREATE FUNCTION 访问权。当允许用户创建存储函数时，请记住，在执行时，存储函数有对数据库对象的拥有者的访问权限。因此，要确保存储函数的拥有者至少有对存储函数内使用的数据库对象的 REFERENCES 访问权。

正如读者将在本技巧中学习的，CREATE FUNCTION 语句的句法有以下 3 种形式之一，具体那种形式，要根据函数返回的数据类型来定。存储函数要么返回一个单一（标量）值，要么返回一个表。

为了创建一个返回标量（而不是一个表）的函数，要用以下句法：

```
CREATE FUNCTION [<owner name>.]<function name>
[([@<parameter name> [AS] <scalar data type>
|=<default value>])...n]]
RETURNS <scalar return data type>
[WITH {ENCRYPTION|SCHEMABINDING}]
[AS]
BEGIN
<Transact-SQL statements>
RETURN <scalar expression>
END
```

其中：

- <owner name>是拥有要创建函数的用户的用户名。在<owner name>处输入的用户名必须是已有的用户 ID。在创建函数时，请记住，存储函数以拥有者的访问权限来操作数据库对象。
- <function name>是存储函数名，可长达 128 个字符。
- <parameter name>是用作存储函数中的变量的参数名。在存储函数调用中必须提供每个参数的值，或者作为 CREATE FUNCTION 语句的一部分将其设置为缺省值。（当用户调用函数时，必须在调用函数中省略了的参数处键入关键词 DEFAULT。）虽然在存储函数中参数可以作为变量使用，但参数不能用作列名、表名和其他数据库对象名。
- <scalar data type>是参数的数据类型。参数可以是任何有效的 SQL 标量数据类型（包括 TEXT、NTEXT 和 IMAGE）。用户定义的数据类型，甚至是标量时也是不允许的。存储函数的“只有标量的数据类型”限制意味着，函数的任何输入参数都不能是表或

者游标。

- `<default value>` 是参数的默认（初始）值。如果指定了此参数，用户可通过指定关键词 `DEFAULT` 代替参数来调用函数。
- `...,n` 指明 `CREATE FUNCTION` 语句最多可以有 2100 个参数。
- `ENCRYPTION` 告诉 MS-SQL Server 对 `SYSCOMMENTS` 表中的存储函数的文本加密，以防止用户查看已编译了的存储函数中的语句。
- `SCHEMABINDING` 告诉 MS-SQL Server 将函数与其引用的数据库对象绑定一起。如果与数据库对象绑定，那么，函数引用的对象便不能（用 `ALTER` 语句）更改或者（用 `DROP` 语句）删除。为了去掉函数的“绑定”，以便更改或删除函数内引用的数据库对象，则必须要么删除函数，要么执行没有指定 `SCHEMABINDING` 选项的 `ALTER FUNCTION` 语句。为了创建带有 `SCHEMABINDING` 的存储函数，以下条件必须为真：任何用户定义的函数以及函数内的视图必须也是架构绑定的；函数引用的对象不能有两部分的名称；存储函数以及它所引用的对象必须在同一数据库内；而且用户执行 `CREATE FUNCTION` 语句时，必须有对所有函数内使用的数据库对象的 `REFERENCES` 权限。
- `<scalar return data type>` 是指存储函数返回其调用者的值的数据类型。由于所有存储函数的参数都是“输入”参数，函数只能返回单一标量值（或者在内嵌表或多语句表函数的情况下可为一个值表，其句法将在以后讨论）。函数返回的值可以是任何合法的 SQL 标量数据类型（包括 `TEXT`、`NTEXT` 和 `IMAGE`）。用户定义的数据类型，即使是标量也是不允许的。
- `<Transact-SQL statements>` 是 Transact-SQL 语句。存储过程的最大容量为 128M。与存储过程不同，存储函数内的语句可能不以任何方式改变在函数之外定义的数据库对象（如全局游标、表和视图等）。

例如，为创建返回特定雇员总销售量的存储函数，可执行如下 `CREATE FUNCTION` 语句：

```
CREATE FUNCTION ufn_employee_total_sales
(@salesrep_ID INTEGER)
RETURNS REAL
BEGIN
RETURN (SELECT SUM(order_total)
FROM cust_orders
WHERE cust_orders.salesrep_ID = @salesrep_ID)
END
```

为了创建内联表值的存储函数——一种能返回一个表的函数，表中的每列都不是在函数的 `RETURNS` 子句中定义的——可使用以下句法：

```
CREATE FUNCTION {<owner name>.<function name>}
([{@<parameter name> [AS] <scalar data type>
[=<default value>]}[...n]})
RETURNS TABLE
[WITH {ENCRYPTION|SCHEMABINDING}]
[AS]
RETURN {() <SELECT statement> []}
```

在这种形式中，在 CREATE FUNCTION 语句的 RETURNS 子句中的关键词 TABLE 表明函数将向主调程序返回一个（数值）表。由于在 RETURNS 子句中并没有表本身的描述，因此，表是在函数结尾处的 RETURN 子句内由 SELECT 语句定义的。

例如，为了创建一个能返回指定给特定销售人员的所有顾客信息的函数，可以使用以下 CREATE FUNCTION 语句：

```
CREATE FUNCTION ufn_employee_customer_list
(@salesrep_ID INTEGER)
RETURNS TABLE
AS
RETURN SELECT * FROM customers
WHERE customers.salesperson = @salesrep_ID
```

CREATE FUNCTION 语句的第 3 种形式也定义了一个能返回表（而不是标量值）的函数。然而，虽然内联函数的表定义仍由 SELECT 语句（在函数的 RETURN 子句内）给出，但多语句表函数的句法是在函数的 RETURNS 子句内包括表定义的，如下所示：

```
CREATE FUNCTION [<owner name>.]<function name>
([{@<parameter name> [AS] <scalar data type>
[=<default value>]}[...n]])
RETURNS @<table name> TABLE <table definition>
[WITH {ENCRYPTION|SCHEMABINDING}]
[AS]
BEGIN
<Transact-SQL statements>
RETURN
END
```

例如，为了创建返回来自 3 个销售办公室（每个办公室都有自己的客户表）的所有顾客列表的函数，可用以下 CREATE FUNCTION 语句：

```
CREATE FUNCTION ufn_composite_customer_list ()
RETURNS @composite_customer_list TABLE
(cust_ID INTEGER,
first_name VARCHAR(30),
last_name VARCHAR(30),
sales_office SMALLINT,
salesperson INTEGER)
AS
BEGIN
/* Build the consolidated list from office 1 customers */

INSERT @composite_customer_list
SELECT cust_ID, first_name, last_name, 1, salesperson
FROM Officel_Customers

/* Add office 2 customers to the consolidated list */

INSERT @composite_customer_list
SELECT cust_ID, first_name, last_name, 2, salesperson
```

```
FROM Office2_Customers

/* Add office 3 customers to the consolidated list */

INSERT @composite_customer_list
SELECT cust_ID, first_name, last_name, 3, salesperson
FROM Office3_Customers

RETURN
END
```

当执行 CREATE FUNCTION 语句时, DBMS 为存储函数中的每个语句进行分析、确认、优化并产生执行计划。如果存储函数中的任何一个语句的句法不正确, 或者试图改变函数外部的数据库对象的结构或其内的数据, DBMS 将拒绝存储函数并报告它所发现的错误。如果由于错误导致 DBMS 不能创建存储函数, 则请改正所报告的错误, 而后重新向 DBMS 提交更新后的 CREATE FUNCTION 语句供执行。如果存储函数中一条或多条语句引用其他还没有实现 (也就是说, 目前尚不存在) 的存储函数, DBMS 将发出警报信息, 但仍将在 MS-SQL Server 上编译并安装存储函数。请注意, 虽然存储函数可调用其他用户定义的和内建的存储函数, 但不能调用用户定义的存储过程。

注意: 存储函数处理语句错误时与存储过程不同。如果一条 Transact-SQL 错误导致 DBMS 停止执行存储过程中的语句, DBMS 继续存储过程中下一个语句的执行。相反, 如果一个错误暂停函数中语句的执行, DBMS 立即返回主调程序, 并停止执行激活存储函数的语句。

技巧 418 使用 MS-SQL Server Enterprise Manager 查看或修改存储过程或函数

如果使用 MS-SQL Server DBMS, 可用以下句法调用系统存储过程 SP_HELPTEXT, 以便显示默认值、规则、视图或未加密的存储函数、过程或触发器:

```
SP_HELPTEXT <object name>
```

例如, 为了显示存储过程 USP_PROCESS_CHECK 的代码 (在技巧 414 “使用存储过程参数返回值” 中描述的), 此时应按如下形式调用 SP_HELPTEXT 存储过程:

```
SP_HELPTEXT usp_process_check
```

如果想修改 (以及查看) 存储过程或用户定义函数, 可使用 MS-SQL Server Enterprise Manager 通过执行以下步骤显示过程或函数:

1. 单击 Windows Start (开始) 按钮, Windows 显示 Start 菜单。
2. 把鼠标移到 Programs (程序), 选择 Microsoft SQL Server 选项, 单击 Enterprise Manager, Windows 就在一个新的应用窗口启动 Enterprise Manager。
3. 单击 SQL Server Group 左边的加号 (+), 显示网络上可用的 MS-SQL Servers 清单。
4. 在有想要显示 (或许想修改) 的存储过程或函数的 SQL Server 服务器的左边单击加号 (+), Enterprise Manager 将显示所选择的 SQL Server 上的 Databases, Data Transformation、Management、Security 和 Support Services 文件夹。
5. 单击 Databases 文件夹的左边的加号 (+), 以显示由 MS-SQL Server 所管理的数据库清单, 然后再单击有想要显示的存储过程或函数的数据库左边的加号 (+)。对于本例来说, 单击 SQL TIPS 文件夹左边的加号 (+)。MS-SQL Server 显示表、视图、存储过程以及其他数据

库所管理的资源的图标。

6. 为了显示该数据库内的存储过程列表，单击 Stored Procedures（存储过程）图标。或者，为了显示存储函数列表，单击 User-Defined Functions（用户定义函数）图标。Enterprise Manager 将用其右窗格显示在第 5 步选择的数据库内的存储过程或函数的列表。

7. 在 Enterprise Managers 右窗格内，右击想要查看或修改的存储过程或函数。Enterprise Manager 显示弹出式菜单，从该菜单中可选择 Properties 命令。对于本例来说，在 USP_PROCESS_CHECK534 上右击，然后从弹出菜单上选择 Properties 命令，以便显示存储过程代码，如图 21.1 所示。

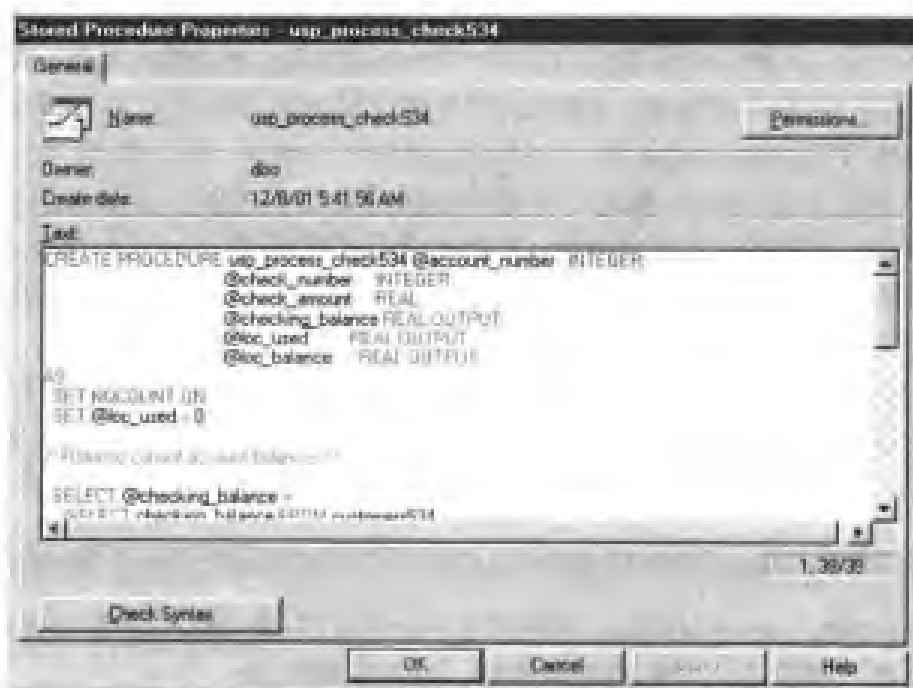


图 21.1 MS-SQL Server Enterprise Manager 的 Procedure Properties 对话框

8. 如果想修改存储过程或函数的行为，可在 Stored Procedure (or User-Defined Function) Properties（存储过程或用户定义函数属性）对话框中的 Text 窗口中按要求修改、删除或添加，然后单击 Check Syntax（检查句法）按钮。

9. 如有必要可重复步骤 8，以修改 Enterprise Manager 对 Text 窗口中的过程或函数的句法检查所报告的错误。

10. 为了更新允许执行存储过程或函数的用户列表，可单击 Permissions（权限）按钮。Enterprise Manager 显示带有 Permissions 选项卡 Object Properties（对象属性）对话框。在允许执行存储过程的用户名右边的复选框内单击使之出现选择标记（为了给予所有用户以执行存储过程的权限，可在 public 角色右边的复选框内单击使之出现选择标记）。然后，单击 OK 按钮，保存权限更新并返回 Stored Procedure (or User-Defined Function) Properties 对话框。

11. 为了保存更改并使得更新的存储过程或函数对 DBMS 可用，单击 OK 按钮。

在完成步骤 11 后，Enterprise Manager 将把存储过程（或用户定义的函数）文本保存到磁盘，并将其作为数据库对象加以保存（通过将存储过程或函数文本插入到服务器的系统表里）。然后，Enterprise Manager 退出 Stored Procedure (or User-Defined Function) Properties 对话框。

为了创建新的存储过程或函数，在步骤 7 的 Enterprise Manager 右窗格内的任何一处空白处单击右键。然后从弹出式菜单上选择 New Stored Procedure（新建存储过程）（或 New User-Defined Function，新建用户定义函数）命令。接着，完成步骤 8~步骤 11，当用户调用步骤 8 中的程序或函数时，请输入想让 DBMS 执行的存储过程（或函数）的标题和语句。

注意：如果在 CREATE PROCEDURE 或 CREATE FUNCTION 语句中，包含有 WITH ENCRYPTION 子句，SP_HELPTEXT 或 Enterprise Manager 都不能显示存储过程（或者用户定义函数）的源代码。此外，在把存储过程或函数保存到磁盘后，不能用 Enterprise Manager 通过编辑存储过程（或用户定义函数）的属性对话框内的文本来改变存储过程或函数的行为。事实上，改变任何一个加密的存储过程或存储函数的惟一方法是删除存储过程或函数，然后重新创建。

技巧 419 使用 Transact-SQL 关键词 DECLARE 和 SELECT 在存储过程中定义变量并为其赋初始值

当执行存储过程或函数内的语句批处理时，常常需要用变量暂时存储数据。正如读者在技巧 416 “在存储过程中使用游标”中所学习的，随着一行一行地处理游标，可以用变量存储变化的总计。除此之外，必须首先把数值从游标行的列中转移到 Transact-SQL 语句可访问的临时变量中，然后才能使用游标行中的数据。

为了声明变量，可使用以下的 DECLARE 语句的句法：

```
DECLARE @<variable name> <data type>
[,...,@<last variable name> <last data type>]
```

这样一来，为了声明存储 INTEGER、REAL 和 DATETIME 值的变量，可写出下列 DECLARE 语句：

```
DECLARE @customer_count INTEGER, @total_sales REAL,
@best_sales_date DATETIME
```

如果愿意，可在同一语句批处理中，写出多个 DECLARE 语句，或者把单条声明语句分成多行，这样就可添加有关每个变量的目的的注释，例如：

```
DECLARE
@customer_count INTEGER /* number of customers serviced
by the salesperson */
@total_sales REAL /* grand total sales made */
@best_sales_date DATETIME /* date most sales made */
```

请注意，变量名可以由任何数目的字母和数字组成，但必须以符号@开始。

为了给变量赋值，可用 SET 或 SELECT 语句。例如，为了给变量赋初始值，语句批处理可包括如下所示的语句：

```
SET @customer_count = 0
SET @total_sales = 0.00
SET @best_sales_date = ''
```

类似地，可用 SELECT 语句设置变量值，如下所示：

```
SELECT @customer_count = 0, @total_sales = 0.00,
@best_sales_date = ''
```

请记住，这里用于初始化变量的 SELECT 语句是标准的 SQL SELECT 语句。同样地，可

写出使用以下句法的 SELECT 语句:

```
SELECT @variable_name = {expression | SELECT statement}
{...{, @<last variable name> =
{expression | SELECT statement}}
[FROM <list of tables>]
[WHERE expression]
[GROUP BY <column names>]
[HAVING expression]
[ORDER BY <column names>]
```

这样一来, 为了打印下了最大定单的顾客的姓名以及销售人员的 ID 和总销售量, 可用下列语句批处理:

```
DECLARE
@customer_name VARCHAR(26)
@largest_order MONEY /* largest order placed by a
single customer */
@total_sales MONEY /* grand total sales made */
@salesperson_ID
SELECT @customer_name =
RTRIM(first_name) + ' ' + RTRIM(last_name),
@largest_order = order_total,
@salesperson_ID = salesperson_ID
FROM customers
ORDER BY order_total

SELECT @total_sales = SUM(order_total)
FROM customers
WHERE salesperson_ID = @salesperson_ID
PRINT 'Largest order ('+CONVERT(VARCHAR(10),@largest_order)
+') placed by'+@customer_name+' Sales Rep: '
+CONVERT(VARCHAR(6),@salesperson_ID)
PRINT 'Total sales for rep: '
+CONVERT(VARCHAR(6),@salesperson_ID)+' = '
+CONVERT(VARCHAR(10),@total_sales)
```

如果用于为变量赋值的 SELECT 语句返回多行 (如本例中的第一条 SELECT 语句就是这种情况), DBMS 用来自返回的结果集中的最后一行为变量赋值。

请注意, 使用 DECLARE 语句创建的变量是“局部”变量。这就意味着在 DBMS 执行语句批处理、过程或函数的最后一个语句之后, 在语句批处理、过程或函数内声明的变量不再可用。

```
DROP PROCEDURE Konrad.usp_increase_my_pay
```


第 22 章 修理及维护 MS-SQL Server 数据库文件

技巧 420 理解 MS-SQL Server 的 Database Consistency Checker (DBCC, 数据库一致检查器)

数据库一致性检查器 (DBCC) 是可以用于获取 MS-SQL Server 管理的有关数据库对象的详细信息的工具。笼统地说, DBCC 是一组语句 (有时称为数据库控制台命令), 可用来确保 DBMS 一切正常。正如读者将在本技巧和技巧 421~技巧 424 中看到的, DBCC 语句检查数据库的物理和逻辑一致性。许多语句不仅能探测, 而且还能修正它所探测到的问题。

通常, 执行 DBCC 语句既可修正 DBMS 内报告的问题 (在求助于数据库恢复之前, 这可能会引用某些数据丢失), 也可在 DBMS “呆滞” 或者存储过程调用看起来产生了错误或不一致的结果时确定问题的来源。

DBCC 语句分为以下 4 类:

- 维护——执行对数据库、数据库中的索引或者数据库保存其中的组成逻辑文件组的物理文件的维护任务。
- 杂项——执行诸如显示某个特定的 DBCC 语句的句法信息、改变 DBMS 处理表数据在内存中存储的方式以及从内存中卸载 DLL (扩展的系统存储过程) 等多项任务。
- 状态——检查事务处理日志内的空闲空间量、打开的事务处理、存储过程缓存中的存储过程等的状态。
- 校验——对数据库或数据库内的系统表、表、索引、目录、组成文件组的文件或者数据库页面分配等进行校验 (并修复)。

在执行 DBCC 维护任务和进行校验操作时, 通常最好在 MS-SQL Server 上活动尽可能少时来执行这些任务。当用户正在进行许多更新和改变时, 如果试图运行一致性检查和修理, 可能收到虚假的错误, 因为许多 DBCC 系统调用要求几乎专用数据库。因此, 如果必须在正在使用的数据库中运行 DBCC 语句, 则要认真检查所报告的警告以及错误信息。看看所报告的错误是否因用户导致的死锁而引起的 (这一点已在技巧 265 “理解死锁以及 DBMS 如何解决死锁” 中学习过有关内容) 或其他由于高级别的 DBMS 活动所导致的定时问题。如果有疑虑, (一定要在采取重大修改措施之前) 在运行 DBCC 维护和校验语句之前, 用系统的存储过程 SP_DBOPTION 将数据库设置为单个用户模式。

例如, 假定想在 SQLTips 数据库上运行一致性检查。首先, 要求所有用户退出, 从而对数据库拥有专有控制权, 然后用以下语句把数据库设置为单个用户模式:

```
sp_dboption SQLTips, 'single user', TRUE
```

注意: 把数据库设置为单个用户模式只是意味着在某个时间只能有一个用户登录到数据库上。因此, 如果调用 SP_DBOPTION 时正在使用的数据库不是改变为单用户模式的数据库, 则要在设置为单个用户模式后, 立即通过执行一条 “USE <database name>” 语句确保自己是惟一允许使用数据库的人。这样, 在以下的例子中, 可利用以前的 SP_DBOPTION 调用并执

行下列 USE 语句：

```
USE SQLTips
```

在运行完 DBCC 检查之后，最好执行任何必要的维护或修改操作。一定要将数据库状态返回到多用户模式，以使用户又能登录数据库。为了把数据库改回到多用户模式，可执行以下语句：

```
sp_dboption SQLTips, 'single user', FALSE
```

当然，在 SP_DBOPTION 调用中，把数据库设置为单个用户模式以及在以后调用同一系统的存储过程，恢复多用户访问时，一定要用自己的数据库名代替其中的 SQLTips。

技巧 421~技巧 424 将提供 4 类 DBCC 语句中的每条语句的句法和讨论。

技巧 421 理解 DBCC 的维护语句

可使用以下 5 条 DBCC 维护选项，以重建并整理索引碎片，压缩数据库文件，并更新空间使用率表：

- DBREINDEX——为数据库内的表重建一个或多个索引。
- INDEXDEFRAG——在特定表或视图上消除集群和二级索引的碎片。
- SHRINKDATABASE——减少组成数据库的物理数据文件的大小。
- SHRINKFILE——通常用来减小单个物理数据库文件或事务处理日志文件的大小。
- UPDATEUSAGE——报告并改正 SYSINDEXES 表中的不正确的地方。

DBCC 的 DBREINDEX 语句的句法如下：

```
DBCC DBREINDEX ([ 'database.owner.<table name>'  
[,<index name> [,<fill factor>]]]) [WITH NO_INFOMSGS]
```

其中：

- database.owner.<table name> 是想要重建带有索引的表的完全合格的表名。不能省略数据库和拥有者而只给出表名。
- <index name> 是想要让 DBMS 重建的索引名。如果省略了 <index name> 或指定为“”，则 DBMS 将重建表的所有索引。
- <fill factor> 是 DBMS 要用来存储数据的每个索引页所占空间的百分比。每当索引页填满（用数据）时，DBMS 必须把该页一分为二。这将影响到性能，因为分页对于系统资源来说是花费较大的。因此，较低的填充因子（百分比）越低，消耗的磁盘空间就越多，但通常 DBMS 必须增加的新页的次数就越少。关于“填充因子”的更多的信息，请参见技巧 122 “理解 MS-SQL Server CREATE INDEX 语句选项”一节。这里提供的 <fill factor> 成为索引中的新的默认的 FILLFACTOR 值。如果提供的 <fill factor> 为 0，DBMS 用最初定义索引时的 FILLFACTOR（如技巧 122 所述）。
- <WITH NO_INFOMSGS> 告诉 DBCC 压缩（suppress）所有通知消息，即那些严重级别从 0~10 的消息。

使用 DBCC DBREINDEX 既可为表重建单独的索引，也可同时重建表的所有索引。当向表内复制成块数据时，能够同时重建所有索引是特别方便的。这就允许自动地创建 PRIMARY KEY 和 UNIQUE 索引，而不必了解表的结构和约束。

此外，DBCC DBREINDEX 允许重建所有索引，而不必编写多个 DROP INDEX 和 CREATE

INDEX 语句。例如，为了重建一个有 50% 的 FILLFACTOR 的 NORTHWIND（示例）数据库内的 PRODUCTS 表的所有索引，可执行以下语句：

```
DBCC DBREINDEX('northwind.dbo.products',' ',50)
```

DBCC INDEXDEFRAG 语句的句法如下：

```
DBCC INDEXDEFRAG (
{<database name>|<database ID>|0}
,{<table name>|<table ID>|'<view name>'|<view ID>}
,{<index name>|<index ID>} ) [WITH NO_INFOMSGS]
```

其中：

- <database name>|<database ID>|0 是想让 DBMS 整理碎片的带索引的数据库名。如这里所示，可通过名称或 ID 号指定数据库，将 ID 号指定为 0，就告诉 DBMS 要索引的表在当前数据库内。
- <table name>|<table ID>|'<view name>'|<view ID> 是想要消除碎片的带有索引的表名或视图名的名称或数字 ID 号。
- <index name>|<index ID> 是 DBMS 要消除碎片的索引名或者 ID 号。
- <WITH NO_INFOMSGS> 告诉 DBCC 压缩 (suppress) 所有通知消息，即那些严重级别从 0~10 的消息。

使用 DBCC INDEXDEFRAG 既可消除集群索引也可消除非集群索引的碎片。DBCC INDEXDEFRAG 执行两种操作：

- 对索引的叶式级别消除碎片，从而使页的物理顺序与叶节点的从左至右的顺序一致，这样就改善了索引扫描的性能。
- 紧缩索引，在从索引中移动数据时考虑到 FILLFACTOR。DBMS 可删除把数据从一个索引移到另一个时所造成的任何“空”页。

由于消除一个大的有碎片索引中的碎片要花很长时间，通过完成的百分比，DBCC INDEXDEFRAG 每隔 5 分钟报告一次进度情况。

可在任何时候中止消除碎片的进程，DBMS 将保留程序已完成的工作。当决定是消除索引碎片还是全部重建索引时，要记住以下内容：

- DBCC INDEXDEFRAG（与 DBCC DBREINDEX 不同）不能长期保持锁定，并因此在对索引消除碎片时，不会阻塞用户进行查询以及更新。
- 消除碎片相对较少的索引中的碎片花的时间要比重建同样索引花的时间少得多。相反，索引的碎片很多时，要消除其碎片花的时间通常要比重建花的时间长得多。
- 消除索引中的碎片总是全部记录的过程。这要比重建同一索引向数据库事务处理日志中插入的记录条目要多。

DBCC SHRINKDATABASE 语句的句法如下：

```
DBCC SHRINKDATABASE ( <database name> [,<target percent>]
[, {NOTRUNCATE|TRUNCATEONLY}] )
```

其中：

- <database name> 是那些想让 DBMS 收缩文件（大小）的数据库名称。DBMS 通过删除每个数据库文件中可用于附加数据的空闲空间来收缩数据库文件。
- <target percent> 是执行“收缩”操作之后，DBMS 在文件之内留下的自由空间的百分比。

- NOTRUNCATE 是导致 DBMS 在数据库文件中留下“自由”空间的 NOTRUNCATE 选项。如果想让 DBCC SHRINKDATABASE 减小数据库文件的大小，可以省略 NOTRUNCATE 并只指定<target percent>。
- TRUNCATEONLY 释放“自由”空间给操作系统使用。如果想在数据库文件内留有一定百分比的自由空间，（如<target percent>所指定的），可省略 TRUNCATEONLY 选项。如果既指定<target percent>，又指定 TRUNCATEONLY，DBMS 忽略<target percent>并截短数据库文件，把每个文件末尾的自由空间返回给操作系统。

如果想在给每个文件留下一定百分数的“空闲”空间的同时，想让 DBMS 减少数据库文件的大小，可执行 DBCC SHRINKDATABASE 语句，指定每个文件自由空间的百分数。例如，假定有一个有两个数据库文件名为 SQLTips 数据库和一个事务处理日志文件。3 个文件中的每一个文件大小为 20M 字节，包含有 12M 字节的数据。执行以下语句，就告诉 DBMS 把每个文件减少到 15M 字节：

```
DBCC SHRINKDATABASE (SQLTips, 20)
```

这样，每个文件就有 12M 字节的数据加上 3M 字节的自由空间（3M 占 15M 字节的 20%）。DBMS 将把每个文件最后 5M 字节里数据移动到最初 15M 字节的空间里，把 20M 字节的文件减少为 15M 字节的文件。

当执行 DBCC SHRINKDATABASE 语句时，如果指定了 NOTRUNCATE 选项，DBMS 仍把数据从每个文件尾部的已分配的（8Kb）页移到文件前边没有分配的（8Kb）页里。然而，文件的大小没有发生变化。在本例中，执行下列语句使 DBMS 把所有数据移动到每个文件内前 12M 字节里，在每个文件的最后，留下 8M 的自由空间：

```
DBCC SHRINKDATABASE (SQLTips, NOTRUNCATE)
```

在执行 DBCC SHRINKDATABASE 语句时，如果指定了 TRUNCATE 选项，DBMS 并不移动文件中的数据。取而代之的只是简单地截去每个文件尾部所有未分配的（8Kb）的页。例如，如果一个 20M 字节的文件有 12M 字节的数据分布在前 17M 字节的空间里，以下语句将能把该文件的大小减小至 17M 字节，向操作系统返回最后 3M 相邻的“自由空间”：

```
DBCC SHRINKDATABASE (SQLTips, TRUNCATEONLY)
```

请记住，DBCC SHRINKDATABASE（与 DBCC SHRINKFILE 不同），将不减少任何小于最小尺寸（使用 CREATE DATABASE 语句在创建时指定的或者后来用 ALTER DATABASE 语句明确地设置的）的数据库文件（不论是数据文件还是事务处理日志文件）的物理尺寸。

DBCC SHRINKFILE 语句的句法如下：

```
DBCC SHRINKFILE ( (<file name>|<file ID>  
{[,<target size>]|[, {EMPTYFILE|NOTRUNCATE|TRUNCATEONLY}]})  
)
```

其中：

- <file name>|<file ID>是想让 DBMS 减少尺寸的文件名。可通过名称或 ID 号来指定文件。
- <target size>是指定了文件的“目标”尺寸（以 MB 计）的整数。例如，如果一个 20M 字节的文件有 12 兆字节的数据，指定<target size>为 14M 字节，DBMS 将把文件最后 6M 字节的数据移到前面 14M 字节的未使用的（8KB）页里，并在 14M 字节处截短。然而，DBMS 不会把文件减少到比所存所有数据要求的空间还小。例如，一个 20M 字节的文件，其中 12M 字节为数据，指定<target size>为 10M 字节，将把文件

的大小减少到 12M 字节（而不是 10M 字节）。

- **EMPTYFILE** 告诉 DBMS 把目标文件内所有数据移到同一文件组中的其他文件里，留下一个可用 **ALTER DATABASE** 语句来删除空文件。在指定了 **EMPTYFILE** 选项之后，MS-SQL Server 将不再向文件中存入任何数据。
- **NOTRUNCATE** 告诉 DBMS 把所有已分配（8Kb）的超过<target size>的页移到<target size>前的未分配（8Kb）页中。如果省略<target size>，DBMS 把所有已分配的（8Kb）页移到文件的前面，这样一来，所有自由空间都留在文件的尾部。当指定了 **NOTRUNCATE** 选项时，可防止 DBMS 向操作系统释放文件内的“自由”空间。由此，文件大小不会改变。
- **TRUNCATEONLY** 告诉 DBMS 把文件收缩到最后分配的限度内。这样，如果 20M 字节的文件有 12M 字节的数据分布在前 15M 字节里，指定 **TRUNCATEONLY** 将把文件尺寸减少到 15M 字节。DBMS 不移动文件中的任何数据。

可用 **DBCC SHRINKFILE** 语句把文件减少到比其最小尺寸（使用 **CREATE DATABASE** 语句在创建时指定的或者后来用 **ALTER DATABASE** 语句明确地设置的）还小。如果把文件减少到最小尺寸之下，则在 **DBCC SHRINKFILE** 语句中指定的<文件尺寸>将成为该文件的新最小尺寸。

DBCC UPDATEUSAGE 语句的句法如下：

```
DBCC UPDATEUSAGE ( {'<database name>'|0}  
[,{'<table name>'|'<view name>'}  
[,{'<index name>'|'<index ID>'}]] )  
[WITH {[COUNT_ROWS[,NO_INFOMSGS]]|NO_INFOMSGS}]
```

其中：

- <database name>|0 是报告正确的文件使用率统计数字的数据库名。0 意味着使用当前的数据库。
- '<table name>|<view name>'是报告正确使用率统计数字的表或索引视图名。
- '<index name>|<index ID>'是报告使用率统计数字的索引名或索引 ID 号。如果不指定<index name>（或者<index ID>），DBMS 就将更新表或视图上所有索引的统计数字。
- **COUNT_ROWS** 告诉 DBMS 在 **SYSINDEXES** 表中更新当前表或者视图内由表或视图的当前行计数指定的行的 **ROWS** 列。如果未指定表或者视图，DBMS 则更新数据库内的每个表或者视图的对应的 **SYSINDEX** 行内的 **ROWS** 列。**COUNT_ROWS** 选项只影响那些 **SYSINDEX** 行中的 **INDID** 列为 0 或 1 的 **ROWS** 列。
- **NO_INFOMSGS** 告诉 DBMS 压缩所有报告信息的显示。

DBCC UPDATEUSAGE 将改正 **SYSINDEXES** 表内那些对应于表、视图和集群索引的 **ROWS**、**USED**、**RESERVED** 和 **DPAGES** 列。对于非集群索引，不保存其大小的信息。

通常，仅当怀疑系统存储过程 **SP_SPACEUSED** 报告的结果有误时，才运行 **DBCC UPDATEUSAGE**。事实上，在返回表、视图或者索引空间使用率信息之前，**SP_SPACEUSED** 接受一个运行 **DBCC UPDATEUSAGE** 的可选参数。如果发现在 **SYSINDEXES** 表内没有错误，**DBCC UPDATEUSAGE** 不返回数据。相反，如果 **DBCC UPDATEUSAGE** 发现错误（它已改正的），语句返回在 **SYSINDEXES** 表更新的行和列的信息——只要没有指定 **NO_INFOMSGS**

选项。

技巧 422 理解 DBCC 的杂项语句

可使用下列 4 种 DBCC 的杂项选项来管理 MS-SQL Server 的内存使用率或者显示任何 DBCC 语句的句法：

- <DLL name> (FREE)——从内存中卸载扩展存储过程的 DLL。
- HELP——返回指定 DBCC 语句的句法。
- PINTABLE——告诉 DBMS 不要从内存中清除用于指定表的数据。
- UNPINTABLE——如果必要，告诉 DBMS 可从高速缓存中清除表页。

DBCC <DLL name> (FREE)语句的句法如下：

```
DBCC <DLL name> (FREE)
```

其中：

- <DLL name>——是想让 DBMS 从内存中删除的扩展存储过程的动态链接库文件名。

当告诉 DBMS 执行扩展存储过程时，DBMS 把 DLL 程序文件装载到内存中。例如，为了执行 XP_SENDMAIL，DBMS 要装载 sqlmap70.dll 文件。任何扩展存储过程装入内存的 DLL，在关闭 MS-SQL Server 之前，仍保留在内存中。为了卸载扩展存储过程的 DLL（释放内存），可执行 DBCC <DLL NAME> (FREE)语句。例如，为了从内存中卸载 sqlmap70.dll，可向 DBMS 提交以下语句：

```
DBCC xp_sendmail (FREE)
```

通过执行 SP_HELPEXTENDEDPROC，可显示当前 MS-SQL Server 上可用的所有扩展存储过程 DLL 文件的清单。

DBCC HELP 语句的句法如下：

```
DBCC HELP ( '<DBCC statement>' | '@<variable name>' | '?' )
```

其中：

- <DBCC statement> | @<variable name> 是引用括起来的实际字符串（'<DBCC statement>'）或者是其中有 DBCC 语句名的变量（@<variable name>），也就是 DBCC 语句中跟在 DBCC 之后的关键词。
- ? 告诉 DBMS 返回所有可获得“帮助”信息的 DBCC 语句的清单。

例如，为了显示 DBCC 语句“DBCC UPDATEUSAGE”的句法，可向 DBMS 提交下面的 DBCC HELP 语句：

```
DBCC HELP ('UPDATEUSAGE')
```

或者，通过执行以下语句批处理，把语句名放到变量中（如@DBCC_STATEMENT 变量中），并用变量代替字符串实际值（'UPDATEUSAGE'）：

```
DECLARE @DBCC_statement SYSNAME  
SET @DBCC_statement = 'UPDATEUSAGE'  
DBCC HELP (@DBCC_statement)
```

DBCC PINTABLE 语句的句法如下：

```
DBCC PINTABLE ( <database ID> , <table ID> )
```

其中：

- <database ID>是带有不可从内存中清除表的数据库 ID 号。为确定数据库的 ID 号，

可按 `SELECT DB_ID(['<数据库名>'])` 格式调用 `DB_ID` 函数。

- `<table ID>` 是不能从内存中清除的表的 ID 号。为确定表的 ID 号码, 可按 `SELECT OBJECT_ID(['<数据库名>.<表名>'])` 格式调用 `OBJECT_ID` 函数。

`DBCC PINTABLE` 不会导致 DBMS 将表数据读入内存中。不管是否可清除, 通常情况下, 在执行 Transact-SQL 语句时, 表的数据被读进服务器的高速缓存中。然而, 当 MS-SQL Server 需要空间读入新页时, DBMS 将不会从一个不可清除表的内存中抹掉缓存页。可用 `DBCC PINTABLE` 把较小的、频繁使用的表读入内存中。通过这种方式, 在 DBMS 首次提取或 (和) 更新表数据时, 把表数据读入内存, 以后对同一表数据的引用将从内存而不是磁盘读取中加以满足。

注意: 在使用 `DBCC PINTABLE` 语句时要特别注意。如果把一个太大的表或太多的表定到内存中, 可能严重破坏系统的性能。通过把特定的表 (或一组表) 定到内存中, 可能使高速缓存不能用于满足对其他表的数据请求。事实上, 如果把一个比高速缓存大得多的表定到内存中, 它可能占满整个高速缓存, 从而使 DBMS 的运行速度大大降低。如果高速缓存被占满, `SYSADMIN` 成员必须关闭 MS-SQL Server, 重新启动 DBMS, 然后用 `DBCC UNPINTABLE` 去取消所定到内存中的表 (或多个表)。

`DBCC UNPINTABLE` 语句的句法如下:

```
DBCC UNPINTABLE ( <database ID>,<table ID> )
```

其中:

- `<database ID>` 是带有那些要被清除的表的数据库 ID 号。为确定数据库 ID 号, 为确定数据库的 ID 号, 可按 `SELECT DB_ID(['<数据库名>'])` 格式调用 `DB_ID` 函数。
- `<table ID>` 是要被清除的表的 ID 号。为确定表 ID 号, 可按 `SELECT OBJECT_ID(['<数据库名>.<表名>'])` 格式调用 `OBJECT_ID` 函数。

在执行 `DBCC UNPINTABLE` 语句时, DBMS 不直接从高速缓存中抹掉表数据。`DBCC UNPINTABLE` 语句只是简单地标记可清除的表, 告诉 DBMS 如果需要空间从磁盘读入新页, DBMS 可以从内存中清除表的数据。

技巧 423 理解 DBCC 的状态语句

可以用以下 6 条 DBCC 状态语句中的前两条来显示最新的每个活动用户连接向 DBMS 发送的语句和并 DBMS 返回的结果集。其余 4 条语句可检查索引和表数据的碎片统计数字, 监视事务处理日志使用率, 并显示当前会话的连接选项设置:

- `INPUTBUFFER`——显示 MS-SQL Server 接收到的来自与 DBMS 连接的特定客户的最后一个语句。
- `OUTPUTBUFFER`——以十六进制和 ASCII 码形式显示发送到与 DBMS 连接的特定客户的最新结果集。
- `SHOWCONTIG`——显示表及其索引的碎片信息。
- `OPENTRAN`——显示有关最老的活动事务处理的信息, 如果有的话, 则为 DBMS 正要加以处理的信息。
- `SQLPERF`——显示 MS-SQL Server 管理的所有数据库的有关事务处理日志的统计数字。
- `USEROPTIONS`——显示当前与 DBMS 连接的用户选项集。

DBCC INPUTBUFFER 语句的句法如下：

```
DBCC INPUTBUFFER ( <system process ID> )
```

其中：

- <system process ID>是用户与 DBMS 连接的 ID 号（SPID）。可调用系统存储过程 SP_WHO 以获取当前运行于 DBMS 上的进程列表。由 SP_WHO 返回的结果集的每行给出系统进程 ID（SPID）、当前状态、用户的登录名、进程正在使用的数据库名以及提交给 DBMS 执行的最新请求名。

为了显示特定连接向 DBMS 提交的最后一条语句的全部文本，首先执行以下存储过程调用，以确定连接的 SPID：

```
SP_WHO
```

请查看 LOGINAME 和 HOSTNAME 列，以便找出想要显示的最后事务处理在 MS-SQL Server 上的用户名所在的行。然后，使用显示在行的 SPID 列的数字，作为 DBCC INPUTBUFFER 语句中的 <system process ID>。DBCC 进程将显示 DBMS 从连接中收到的最后一条语句。例如，为了显示从用户连接 13 中收到的最后一条语句，可提交下面的语句：

```
DBCC INPUTBUFFER(13)
```

DBCC OUTPUTBUFFER 语句的句法如下：

```
DBCC OUTPUTBUFFER ( <system process ID> )
```

其中：

- <system process ID>是用户与 DBMS 连接的 ID 号（SPID）。可调用系统存储过程 SP_WHO 以获取当前运行于 DBMS 上的进程列表。由 SP_WHO 返回的结果集的每行给出系统进程 ID（SPID）、当前状态、用户的登录名、进程正在使用的数据库名以及提交给 DBMS 执行的最新请求名。

为了显示 DBMS 向连接号为 20 的用户发送最新的结果集，例如可向 DBMS 提交以下语句：

```
DBCC OUTPUTBUFFER (20)
```

DBMS 将显示由 DBMS 最后发给连接号为 20 的客户端的结果集，其中有结果集中的每个字符。表内的每一行在左边显示所发送字符的十六进制值，而在每行的右侧提供那些字符的 ASCII 码（如果有的话）。

DBCC SHOWCONTIG 语句的句法如下：

```
DBCC SHOWCONTIG
```

```
[ ( {<table name>|<table ID>|<view name>|<view ID>} ]
```

```
[, <index name>|<index ID>] ) ]
```

```
[ WITH {ALL_INDEXES|FAST[, ALLINDEXES]} ]
```

```
TABLERESULTS [ , {ALL_INDEXES} ] [ , {FAST|ALL_LEVELS} ] ) ]
```

其中：

- <table name>|<table ID>|<view name>|<view ID>是要报告碎片数据的表或索引的视图名。（可通过名称或 ID 号来指定表或索引的视图）。如果既不指明表也不指明视图名或者 ID，DBMS 将检查并报告数据库内所有表和索引视图的碎片信息。
- <index name>|<index ID>是要报告碎片统计数字的索引名或其 ID 号。如果不指明 <index name> 或 <index ID>，DBMS 将报告指定表或视图上的基索引（也就是用于 PRIMARY KEY 的）的碎片统计数字。

- FAST 告诉 DBCC 进程执行快速扫描并报告最小量的信息。在执行快速扫描时，DBMS 并不读取索引内的“叶”或数据级的页面。
- TABLERESULTS 告诉 DBCC 进程以表的形式而不是摘要执行的形式显示碎片信息。除了改变输出格式（从报告变为表）之外，如果不指明 TABLERESULTS 选项，那么在结果表中每行包括表名、索引名、记录大小、页数、磁盘限量数据，此外还有返回的所有碎片数据。
- ALL_INDEXES 告诉 DBCC 显示一个表的有关所有索引的碎片信息，即使用<index name>或<index ID>指定了特定的索引也是如此。
- ALL_LEVELS 告诉 DBCC 为每个处理的索引的索引树内的每一级生成输出行。如果指定了 TABLERESULTS 选项却不指定 FAST 选项，则可只指定 ALL_LEVELS 选项。当未指明 ALL_LEVELS 时，DBMS 只处理表数据和索引树内最高级别，即叶级。

由于用户一遍一遍地执行 INSERT、UPDATE 和 DELETE 语句导致未用的（8Kb）页面出现在部分充满的组成索引和表数据文件的页面间，从而使索引和表文件产生碎片。随着数据分布在多页上，于是数据不再能紧缩存储在尽可能少数目的（8Kb）页面内。由于碎片数据或索引文件导致 DBMS 为了提取同样数量的数据必须执行越来越多的页提取，从而使 DBMS 性能下降。

通过执行带有 NOTRUNCATE 选项（正如读者已在技巧 421 “理解 DBCC 的维护语句”中所学习的）的 DBCC SHRINKDATABASE 语句，可以消除表数据的碎片。为了消除索引的碎片，既可以用 DBCC REINDEX 语句重建索引，也可以通过执行 DBCC INDEX DEFRAG 语句消除其中的碎片（不必重建索引）（读者已经在技巧 421 中学习了这两条语句）。

为了提取所需的信息，以便确定数据库内所有索引和表的碎片级别，可执行不带参数的 DBCC SHOWCONTIG 语句：

```
DBCC SHOWCONTIG WITH ALL_INDEXES
```

如果只想检查数据库内一个表的碎片，只需提供表名。例如，为了检查 NORTHWIND 数据库内的 ORDER DETAILS 表的碎片级别，可执行下列语句批处理以便产生如图 22.1 所示的报告：

```
USE NORTHWIND
```

```
DBCC SHOWCONTIG('order details') WITH ALL_INDEXES
```

请注意，可用单引号把表（或者视图）名括起来。事实上，如果名称由多个用空格分开的单词组成（如本例中表名 ORDER DETAILS），则必须用引号把表名括起来。

如图 22.1 所示，由 DBCC SHOWCONTIG 语句返回的碎片数据由下列各项组成：

- Pages Scanned（扫描的页）——表或索引所包括的（8Kb）页面数（TABLERESULTS 列，PAGES）。
- Extents Scanned（扫描的扇区）——表或索引所包括的磁盘扇区的数目。操作系统在向磁盘文件增加空间时一次添加扇区那么多字节。操作系统格式化磁盘时，可设置扇区的尺寸。对于运行于 Windows NT、NTFS 文件系统下的 MS-SQL Server，其最佳扇区尺寸为 64K（TABLERESULTS 列，EXTENTS）。
- Extent Switches（扇区切换）——在表或索引内跨越（8Kb）页面时，DBCC 进程从一个扇区移动到另一扇区的次数（TABLERESULTS 列，EXTENTSWITCHES）。

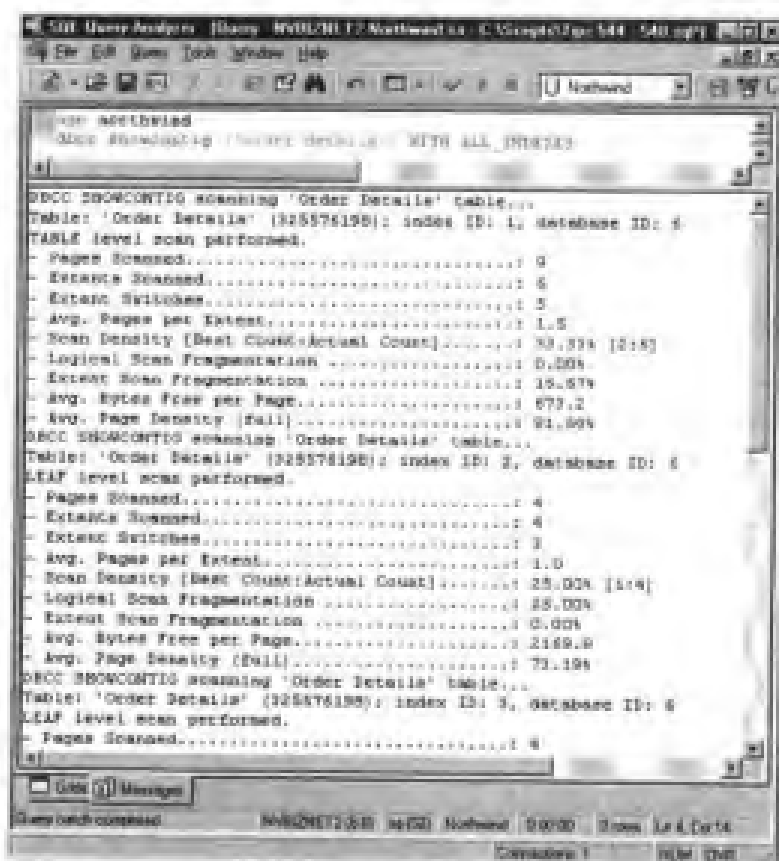


图 22.1 从 MS-SQL Server 的 DBCC SHOWCONTIG 语句产生的输出

- Avg. Pages per Extent（每扇区平均页数）——在每 64K 磁盘扇区中的（8Kb）页面的平均数。
- Scan Density (Best Count, Actual Count)（扫描密度[最佳计数、实际计数]）——Best Count（最佳计数）是指，如果所有扇区都连续地链接在一起的话（也就是，如果数据文件或索引内没有碎片），DBCC 进程遇到的扇区变化的数目。Actual Count（实际计数）是当处理表或索引时，DBCC 进程遇到的扇区变化的实际数目。Scan Density（扫描密度）是 Actual Count 被 Best Count 除的结果，以百分比表示。如果扫描密度为 100%，所有扇区都链接在一起（也就是说，没有碎片）；如果扫描密度小于 100%，就说明有碎片（TABLERESULTS 列，SCANDENSITY、BESTCOUNT 和 ACTUALCOUNT）。
- Logical Scan Fragmentation（逻辑扫描碎片）——当 IAM（索引访问方法）中指明的下一页与索引叶页面中的下一页指针所指向的（8Kb）页面不同的次数的百分比。（TABLERESULTS 列，LOGICALFRAGMENTATION。）
- Extent Scan Fragmentation（扇区扫描碎片）——带有索引的当前（8 Kb）页面的磁盘扇区在物理上不是索引的前一（8 Kb）页面之后的下一扇区的次数百分比（TABLERESULTS 列，EXTENTFRAGMENTATION）。
- Avg. Bytes Free per Page（每页平均自由字节数）——在扫描过的（8 Kb）页面上的自由（未用）的字节平均数目。虽然每页的未使用的空间越小越好，但是行的尺寸可影响自由空间量，行的尺寸越大，那么每（8Kb）页的未使用的空间就越多。

(TABLERESULTS 列, AVERAGEFREEBYTES)。

- Avg. Page Density (full) (平均页密度[满])——以百分数表示的考虑到表行尺寸的页密度。由此, 这个指标是表或索引中的 (8Kb) 页的填满程序的更准确地表示。

(TABLERESULTS 列, AVERAGEPAGEDENSITY。)

在决定是对表还是索引消除碎片时, 请:

- 检查 Avg. Page Density(full), 此指标应该是较高的百分比。
- 检查 Logical Scan Fragmentation 和 Extent Scan Fragmentation, 这两个数字应尽可能地接近 0 (0~10 是可接受范围)。请记住, 如果索引跨越多个文件, Extent Scan Fragmentation 将会较高。
- 比较 Extent Switches 和 Extents Scanned——理想地, 这两个数字将是相等的。请注意, 对于跨越多个文件的索引, 碎片指示数据不起作用。
- 检查 Scan Density——它应尽可能地大。

如果确定一个表或者其索引有碎片, 可使用在技巧 421 “理解 DBCC 维护语句”中所学习的方法, 或者重建或者对索引进行消除碎片并把未用页移到表数据文件的末尾。

DBCC SQLPERF 语句的句法如下:

```
DBCC SQLPERF (LOGSPACE)
```

执行时, DBCC SQLPERF(LOGSPACE)返回一个表, 表行包括显示由 DBMS 所管理的每个数据库名、事务处理日志文件尺寸 (以 MB 为单位) 以及由事务处理数据当前占用的事务处理日志文件的百分比。

DBCC OPENTRAN 语句的句法如下:

```
DBCC OPENTRAN ( ('<database name>' | <database ID> )  
[WITH TABLERESULTS [, NO_INFOMSGS]]
```

其中:

- <database name>|<database ID>是想要查询其事务处理日志以便列出还没有永久地写入数据库的事务处理的数据库名 (或数字 ID)。
- <WITH TABLERESULTS>告诉 DBMS 以表格形式格式化结果集, 以便容易地装入表中。如果未指定 TABLERESULTS 选项, DBMS 返回结果集为 “可读” 的报告而不是表格形式。
- <NO_INFOMSGS>告诉 DBMS 压缩所有报告信息。

如果既不指定 <database name> 也不指定 <database ID>, DBMS 将假定想要显示当前数据库的事务处理日志中未提交的语句。可使 DBCC OPENTRAN 清除未提交的事务处理, 而不必让所有用户注销, 或者关闭后再重新启动 MS-SQL Server。如不这样, 可使用 DBCC OPENTRAN 来确定哪个用户连接有未提交的事务处理。然后, 既可让每个有打开事务处理的用户要么提交 COMMIT 语句, 要么提交 ROLLBACK 语句以关闭事务处理, 又可使用 SP_WHO 的存储过程调用的输出来确定用户的 SPID, 如果需要, 可终止连接。

DBCC USEROPTIONS 语句的句法如下:

```
DBCC USEROPTIONS
```

执行时, DBCC USEROPTIONS 返回一个如图 22.2 所示的表, 其中为每个会话选项集及其当前值包括一行信息。

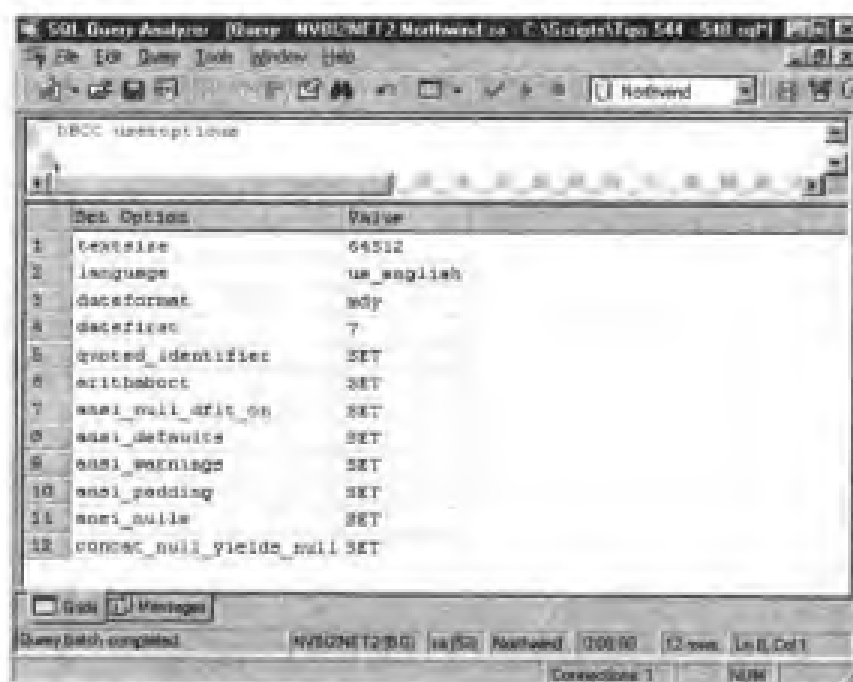


图 22.2 从 MS-SQL Server 的 DBCC USEROPTIONS 语句中返回的结果集

技巧 424 理解 DBCC 的确认语句

可以使用以下 7 条 DBCC 确认语句检查或改正数据库文件中存在的问题。使用 DBCC 确认语句，可检查磁盘分配结构完整性、数据库表、索引和视图的完整性以及组成系统目录的表中数据的一致性。还可检查违反约束情况以及设定表的 IDENTITY 属性值。除了报告所发现的问题之外，大多数 DBCC 语句都有“修理”选项，在单用户模式下执行时可用于改正错误：

- CHECKALLOC——检查（也可修理）指定数据库的磁盘分配结构。
- CHECKCATALOG——检查数据库内系统表间和表内的一致性。
- CHECKCONSTRAINTS——检查数据库内一个或所有表上的一个或所有 FOREIGN KEY 和 CHECK 约束的完整性。
- CHECKDB——检查（也可修理）磁盘分配结构以及数据库内所有对象的完整性。
- CHECKFILEGROUP——检查（也可修理）文件组内所有表的磁盘分配和结构完整性。
- CHECKIDENT——检查并允许为有 IDENTITY 列的表设定 IDENTITY 属性值。
- CHECKTABLE——检查（也可修理）表中数据的完整性。

DBCC CHECKALLOC 语句的句法如下：

```
DBCC CHECKALLOC ( '<database name>'
[, {REPAIR_ALLOW_DATA_LOSS|REPAIR_FAST|REPAIR_REBUILD}] )
[WITH
{[ALL_ERRORMSG|NOINFOMSG] [, {ESTIMATEONLY}]] ]
```

其中：

- <database name> 是想要让 DBCC 检查分配错误的数据库名。当带 REPAIR_ALLOW_DATA_LOSS 选项运行时，DBCC CHECKALLOC 将确保索引结构（告诉 DBMS 在磁盘的何处可找到数据库文件内的数据）实际指向表或索引数据，还要确保驻留有数据的磁盘扇区被分配到数据库文件内的数据库对象存储区。

- REPAIR_FAST 告诉 DBCC 进程执行由 REPAIR_FAST 选项指定的较小的、不太费时的修正，如删除非集群索引中额外的键字。在 REPAIR_FAST 选项下执行的修正将不会导致数据丢失。
- REPAIR_REBUILD 告诉 DBCC 进程执行与由 REPAIR_FAST 选项指定的相同的修理任务，但加上较费时的修理，例如重建索引。在 REPAIR_REBUILD 选项下执行的修正不会引起数据丢失。
- REPAIR_ALLOW_DATA_LOSS 告诉 DBCC 进程执行与指定 REPAIR_REBUILD 选项同样的修理任务，但加上修正分配错误、结构行错误和删除非法的文本对象。在 REPAIR_ALLOW_DATA_LOSS 选项下执行修正时，如果 DBCC 进程必须删除一些被损坏的数据，（顾名思义）则可导致某些数据丢失。
- ALL_ERRORMSGs 告诉 DBCC 进程显示所有错误信息。如果指明 NO_INFOMSGs 而没有指明 ALL_ERRORMSGs，DBCC CHECKALLOC 将只显示每个对象的前 200 条错误信息。
- NO_INFOMSGs 告诉 DBCC 进程压缩所有报告的信息（并只显示错误信息）。
- ESTIMATE_ONLY 当执行带此选项和指定参数执行 DBCC CHECKALLOC 语句时，告诉 DBCC 进程显示估计的 DBCC CHECKALLOC 语句所需的 TEMPDB 内的空间量。

注意：执行带有修理选项（REPAIR_FAST、REPAIR_REBUILD 或者 REPAIR_ALLOW_DATA_LOSS）之一的 DBCC CHECKALLOC 语句时，必须首先把要修理的数据库设定为单用户模式。

DBCC CHECKCATALOG 语句的句法如下：

```
DBCC CHECKCATALOG ( '<database name>' ) [WITH NO_INFOMSGS]
```

其中：

- <database name>是想让 DBCC 检查系统表内数据一致性的数据库名。
- NO_INFOMSGs 告诉 DBCC 进程在报告由系统目录使用的空间时压缩到通知信息。如果指定 NO_INFOMSGs 选项，DBCC CHECKCATALOG 将只显示进程遇到的前 200 个错误信息。为了显示所有错误消息，应忽略 DBCC CHECKCATALOG 语句的 NO_INFOMSGs 选项。

在执行时，DBCC CHECKCATALOG 做诸如确保 SYSCOLUMNS 表内的每种数据类型在 SYSTYPES 表内都有匹配的条目以及列在 SYSOBJECTS 表中的每个视图在 SYSCOLUMNS 表中都至少有一列信息这样的事情。

DBCC CHECKCONSTRAINTS 语句的句法如下：

```
DBCC CHECKCONSTRAINTS  
[( '<table name>' | '<constraint name>' )]  
[WITH {ALL_ERRORMSGs | ALL_CONSTRAINTS}]
```

其中：

- <table name>是想让 DBCC 进程检查的带有约束的表名。如果指定表名，DBCC 检查表的所有启用的约束。
- <constraint name>是想让 DBCC 检查的约束名。
- ALL_CONSTRAINTS 告诉 DBCC 检查由表名指定的表上的所有约束（既包括启用的

也包括禁用的)。或者,如果既未指明<table name>,也未指明<constraint name>,则告诉 DBCC 进程检查所有表上的所有约束(如果指定了<constraint name>,则 ALL_CONSTRAINTS 不起作用)。

- ALL_ERRORMSGS 告诉 DBCC 进程显示所有违反所检查约束的行。如果省略了 ALL_ERRORMSGS 选项, DBCC 进程将只报告所检查的每个表中遇到的前 200 个违反约束的情况。

如果既没有指明 <table name> 也没有指明 <constraint name>, DBCC CHECKCONSTRAINTS 语句将检查当前数据库内所有表上的所有启用的约束。

对于此语句发现的每个违反 FOREIGN KEY 和 CHECK 约束的情况, DBCC CHECKCONSTRAINTS 返回表名、约束名以及导致约束违反的数据值。

DBCC CHECKDB 语句的句法如下:

```
DBCC CHECKDB ( '<database name>'
[, NOINDEX | { REPAIR_ALLOW_DATA_LOSS | REPAIR_FAST |
REPAIR_REBUILD } ] )
[ WITH { [ ALL_ERRORMSGS ] [ , [ NO_INFOMSGS ] ] [ , [ TABLOCK ] ]
[ , [ ESTIMATEONLY ] ] [ , [ PHYSICAL_ONLY ] ] } ]
```

其中:

- <database name> 是想让 DBCC 检查分配和结构完整性错误的数据库名。
- NOINDEX 指定不检查非系统表(也就是用户定义的)上的非集群索引。
- REPAIR_FAST 告诉 DBCC 进程执行较小的、不费时的修理,如删除非集群索引中多余的键字。在 REPAIR_FAST 选项下执行修理,不会丢失任何数据。
- REPAIR_REBUILD 告诉 DBCC 进程执行与由 REPAIR_FAST 选项指定的相同的修理任务,但加上较费时的修理,例如重建索引。在 REPAIR_REBUILD 选项下执行的修正不会引起数据丢失。
- REPAIR_ALLOW_DATA_LOSS 告诉 DBCC 进程执行与指定 REPAIR_REBUILD 选项同样的修理任务,但加上修正分配错误、结构行错误和删除非法的文本对象。在 REPAIR_ALLOW_DATA_LOSS 选项下执行修正时,如果 DBCC 进程必须删除一些被损坏的数据,(顾名思义)则可导致某些数据丢失。
- ALL_ERRORMSGS 告诉 DBCC 进程显示所有错误信息。如果指明 NO_INFOMSGS 而没有指明 ALL_ERRORMSGS, DBCC CHECKALLOC 将只显示每个对象的前 200 条错误信息。
- NO_INFOMSGS 告诉 DBCC 进程压缩所有报告的信息(并只显示错误信息)。
- TABLOCK 告诉 DBCC 获取共享的表锁定(而不是与行或页面锁定)。指定 TABLOCK 选项使 DBCC CHECKDB 在有很重负载的 DBMS 上运行得更快。然而,发出表锁定将降低并发性,因为用户被强制等待 DBCC 进程完成其在整个表(而不是在单行或行的 8Kb 页)上所进行的工作。
- ESTIMATE_ONLY 当执行带此选项和指定参数执行 DBCC CHECKALLOC 语句时,告诉 DBCC 进程显示估计的 DBCC CHECKALLOC 语句所需的 TEMPDB 内的空间量。
- PHYSICAL_ONLY 告诉 DBCC 进程通过检查页和记录头部的结构、数据库文件分配结构以及为可导致损坏数据的硬件故障而分配的磁盘扇区,从而只检查数据库的物理

一致性。PHYSICAL_ONLY 意味着想要的选项是 NO_INFOMSGS, 不能与 3 个修理选项一起 (REPAIR_FAST、REPAIR_REBUILD 或 REPAIR_ALLOW_DATA_LOSS) 指定 PHYSICAL_ONLY 选项。

注意: 执行带有修正选项 (REPAIR_FAST、REPAIR_REBUILD 或者 REPAIR_ALLOW_DATA_LOSS) 之一的 DBCC CHECKDB, 必须首先把要修正的数据库设定为单用户模式。

在执行时如果没有 PHYSICAL ONLY 选项, DBCC CHECKDB 是最全面的 DBCC 确认语句, 因为可确定并修理最大可能范围内的错误。简而言之, DBCC CHECKDB 检验数据库内所有内容的完整性。因此, 如果运行 DBCC CHECKDB, 不必运行 DBCC CHECKALLOC 或 DBCC CHECKTABLE, 因为 DBCC CHECKDB 执行了那些语句所提供的所有检验和修理操作。

DBCC CHECKFILEGROUP 语句的句法如下:

```
DBCC CHECKFILEGROUP ( [ ('<filegroup name>' | <filegroup ID>)]  
[, NOINDEX] )  
[WITH { [ALL_ERRORMSG] [, [NO_INFOMSGS]] [, [TABLOCK]]  
[, [ESTIMATEONLY]] } ]
```

其中:

- <filegroup name>是想检查表分配和结构完整性的文件组名 (读者已经在技巧 391 “向已有数据库添加文件和文件组” 中学习了如何将数据库对象放入文件组内)。
- <filegroup ID>是分配给数据库内每个文件组的惟一的整数 ID 号。为确定文件组 ID, 首先在 SYSFILEGROUPS 表的 GROUPNAME 列内找到文件组名。然后, 在文件组 ID 行中的 GROUPID 列里找出整数。
- NOINDEX 指定不检查非系统表 (也就是用户定义的) 上的非集群索引。(当运行在 PRIMARY 文件组之上时, DBCC CHECKFILEGROUP 总是检查系统表中的所有索引。)
- ALL_ERRORMSG 告诉 DBCC 进程显示所有错误信息。如果指明 NO_INFOMSGS 而没有指明 ALL_ERRORMSG, DBCC CHECKFILEGROUP 将只显示每个表中的前 200 条错误信息。
- NO_INFOMSGS 告诉 DBCC 进程压缩所有报告的信息 (并只显示错误信息)。
- TABLOCK 告诉 DBCC 进程, 在检查 (和修理) 表的同时共享的表锁定 (而不是与行或页面锁定)。指明 TABLOCK 选项使 BCC CHECKFILEGROUP 在处理任务很重的 DBMS 运行得较快。然而, 发出表锁定将降低并发性, 因为用户被强制等待 DBCC 进程完成其在整个表 (而不是在单行或行的 8Kb 页) 上所进行的工作。
- ESTIMATE_ONLY 当执行带此选项和指定参数执行 DBCC CHECKFILEGROUP 语句时, 告诉 DBCC 进程显示估计的 DBCC CHECKFILEGROUP 语句所需的 TEMPDB 内的空间量。让 DBCC 程序显示在 TEMPDB 中的空间大小, 当执行指明的选项和参数时, DBCC CHECKFILEGROUP 语句需要。

DBCC CHECKFILEGROUP 和 DBCC CHECKDB 类似的地方在于, 二者都检查数据库内表分配和结构的完整性。然而, 虽然 DBCC CHECKDB 检查所有表的完整性时, 但 DBCC CHECKFILEGROUP 只检查那些位于由 <filegroup name> (或 <filegroup ID>) 参数指明的文件组内的表或在此文件组内有索引的表的完整性。

DBCC CHECKIDENT 语句的句法如下:

```
DBCC CHECKIDENT ( '<table name>'
[, {NORESEED|RESEED[, <new reseed value>}] ] )
```

其中:

- <table name>带有想要检查或改变 IDENTITY 属性列的表名（读者已在技巧 17“理解 MS-SQL Server 的 IDENTITY 属性”中学过如何使用 IDENTITY 属性让 DBMS 每在表内插入一行时在表列中插入增加的非 NULL 值）。
- NORESEED 指明 DBCC 进程应报告但不改变 IDENTITY 属性的值。
- RESEED 指明 DBCC 进程应改正 IDENTITY 属性值。如果 IDENTITY 属性小于表的 IDENTITY 列的最高值，DBCC CHECKIDENT 把 IDENTITY 属性设为列内最大值。相反，如果 IDENTITY 属性的值大于或等于表的 IDENTITY 列的最大值，DBCC CHECKIDENT 将不改变 IDENTITY 属性的值。
- <new reseed value>是 DBCC 进程要设定的表的 IDENTITY 属性值。如果指定，<new reseed value>必须跟在 RESEED 选项之后。

没有内在的要求 IDENTITY 列内的值必须惟一。然而，通常人们使用 IDENTITY 属性让 DBMS 向 PRIMARY KEY 或是向带有 UNIQUE 约束的列中插入非 NULL 的增加值。在这样做时，就告诉 DBMS 插入一个惟一值，通过该值可确定插入表内的每一行。因此，在用 DBCC CHECKIDENT 语句改变 IDENTITY 属性时一定要小心。如果表的 IDENTITY 列是 PRIMARY KEY 或受 UNIQUE 约束，则必须把 IDENTITY 属性设定为等于或高于列内已有的最大值。如果不这样做，DBMS 将在某处生成重复的 IDENTITY（种子）值，从而防止用户在表中插入另外的行。

为了报告 IDENTITY 属性的值以及表的 IDENTITY 列的最大值——而不改变 IDENTITY 属性的值，可执行以下 DBCC CHECKIDENT 语句：

```
DBCC CHECKIDENT ('<table name>', NORESEED)
```

为了报告 IDENTITY 属性的值，可执行以下 DBCC CHECKIDENT 语句：

```
DBCC CHECKIDENT ('<table name>', RESEED)
```

如果 IDENTITY 属性的值小于表的 IDENTITY 列的最大值，可按本例中显示的格式执行 DBCC CHECKIDENT 语句，告诉 DBMS 把属性值设定为列内的最大值。

为了报告 IDENTITY 属性的当前值，并将其设定为新值（也就是设置为 <new reseed value>）而不管表的 IDENTITY 列内的最大值，可执行以下 DBCC CHECKIDENT 语句：

```
DBCC CHECKIDENT ('<table name>', RESEED, <new reseed value>)
```

如果表的 IDENTITY 属性的值大于表的 IDENTITY 列的最大值，而且想把 IDENTITY 属性设定为 IDENTITY 列内的最大值，可执行以下语句批处理：

```
DBCC CHECKIDENT ('<table name>', RESEED, 0)
```

```
DBCC CHECKIDENT ('<table name>', RESEED)
```

当然，应该使用带有想要调节 IDENTITY 属性的表名来代替上面语句中的 <table name>，而用 IDENTITY 属性的实际值代替前述语句中的 <new reseed value>。

DBCC CHECKTABLE 语句的句法为：

```
DBCC CHECKTABLE ( '<table name>' | '<view name>'
[, NOINDEX | <index ID> |
{REPAIR_ALLOW_DATA_LOSS | REPAIR_FAST | REPAIR_REBUILD} ] )
```



```
{WITH {[ALL_ERRORMSGSGS]}[, [NO_INFOMSGSGS]}[, [TABLOCK]]  
[, [ESTIMATEONLY]}[, [PHYSICAL_ONLY]] }
```

其中:

- <table name>|<view name>是想让 DBCC 进程检查结构完整性错误的表或有索引的视图名。(虽然 DBCC CHECKTABLE 检查 TEXT、NTEXT 和 IMAGE 页的链接性与尺寸,但并不检查数据库内所有分配结构的一致性。必须运行 DBCC CHECKALLOC 或 DBCC CHECKDB 加以确认。)
- NOINDEX 指定不检查非系统表(也就是用户定义的)上的非集群索引。
- <索引 ID>命令 DBCC 程序检查表以及某个具体索引的完整性(由<索引 ID>给出)。
- REPAIR_FAST 告诉 DBCC 进程执行较小的、不费时的修理,如删除非集群索引中多余的键字。在 REPAIR_FAST 选项下执行修理,不会丢失任何数据。
- REPAIR_REBUILD 告诉 DBCC 进程执行与由 REPAIR_FAST 选项指定的相同的修理任务,但加上较费时的修理,例如重建索引。在 REPAIR_REBUILD 选项下执行的修正不会引起数据丢失。
- REPAIR_ALLOW_DATA_LOSS 告诉 DBCC 进程执行与指定 REPAIR_REBUILD 选项同样的修理任务,但加上修正分配错误、结构行错误和删除非法的文本对象。在 REPAIR_ALLOW_DATA_LOSS 选项下执行修正时,如果 DBCC 进程必须删除一些被损坏的数据,(顾名思义)则可导致某些数据丢失。
- ALL_ERRORMSGSGS 告诉 DBCC 进程显示所有错误信息。如果指明 NO_INFOMSGSGS 而没有指明 ALL_ERRORMSGSGS, DBCC CHECKTABLE 将只显示每个对象的前 200 条错误信息。
- NO_INFOMSGSGS 告诉 DBCC 进程压缩所有报告的信息(并只显示错误信息)。
- TABLOCK 告诉 DBCC 获取共享的表锁定(而不是与行或页面锁定)。指定 TABLOCK 选项使 DBCC CHECKTABLE 在有很重负载的 DBMS 上运行得更快。然而,发出表锁定将降低并发性,因为用户被强制等待 DBCC 进程完成其在整个表(而不是在单行或行的 8Kb 页)上所进行的工作。告诉 DBCC 进程压缩所有报告的信息(并只显示错误信息)。
- ESTIMATE_ONLY 当执行带此选项和指定参数执行 DBCC CHECKTABLE 语句时,告诉 DBCC 进程显示估计的 DBCC CHECKTABLE 语句所需的 TEMPDB 内的空间量。
- PHYSICAL_ONLY 告诉 DBCC 进程通过检查页和记录头部的结构、数据库文件分配结构以及为可导致损坏数据的硬件故障而分配的磁盘扇区,从而只检查数据库的物理一致性。PHYSICAL_ONLY 意味着想要的选项是 NO_INFOMSGSGS,不能与 3 个修理选项一起(REPAIR_FAST、REPAIR_REBUILD 或 REPAIR_ALLOW_DATA_LOSS)指定 PHYSICAL_ONLY 选项。

注意: 执行带有修正选项(REPAIR_FAST、REPAIR_REBUILD 或 REPAIR_ALLOW_DATA_LOSS)之一的 DBCC CHECKTABLE, 必须首先把要修正的数据库设定为单用户模式。

如果想检查数据库内所有表的一致性,可用 DBCC CHECKDB 而不要使用 DBCC CHECKTABLE。

第 23 章 编写高级查询及子查询

技巧 425 理解对用作比较运算符判式的子查询的限制

许多查询使用 WHERE 子句中的比较运算符选择行，在这些行中的一列或多列有特定值或值范围。例如，为了显示 2002 年 2 月 1 日后雇用的雇员名单，可使用以下查询：

```
SELECT first_name, last_name
FROM employees
WHERE hire_date > '2/1/2002'
```

类似地，如要列出来自 California 的所有学生的化学成绩，可使用以下查询：

```
SELECT first_name, last_name, grade
FROM students AS s, grades AS g
WHERE s.home_state = 'CA'
      AND s.student_ID = g.student_ID
      AND g.class = 'chemistry'
```

虽然这两个查询的实质不同，但 WHERE 子句类似的地方在于，这两个子句紧挨在一对标量之间都有比较运算符。事实上，比较运算符（=、<>、>、<、>=或<=）只能用于比较两个标量值——运算符左边的标量值与运算符右边的标量值相比较。然而，与前面两个例子中所做的比较不同，当编写查询时，并不总是知道想用来比较的标量值。幸运的是，可使用子查询（既可是相关的也可是不相关的）来代替比较运算符一边或两边的标量值。

对使用子查询代替标量值的限制是，子查询必须至多返回一个值。也就是说，子查询中的 SELECT 语句必须返回由只有一列的零行或一行组成的结果集。SQL 程序员常常将返回一个标量值的子查询称为标量子查询。

为了列出那些 GPA 比平均 GPA 多 25% 的学生清单，可写出以下查询：

```
SELECT first_name, last_name, gpa
FROM students
WHERE gpa > ((SELECT AVG(gpa) FROM students) * 1.25)
```

如果不知道平均的 GPA，仍可使用一个不相关的标量子查询（跟在大于比较运算符[>]后的），编写根据 DBMS 执行该查询时计算的值得选择行的查询。由于本例中的子查询是不相关的，DBMS 只需执行一次。不相关的子查询，不论在查询中调用多少次，都只返回相同结果集。在本例中，在扫描列在 FROM 子句中的表（STUDENTS）时，优化器将其减少为一个常数。

不要只局限于使用带比较运算符的不相关子查询。虽然处理过程有点复杂（对于 DBMS 来说），但也可用相关子查询。例如，假定你想列出那些以平均\$500,000 或更高的价格卖出房子的房地产经纪人清单，可写出以下查询：

```
SELECT first_name, last_name
FROM realtors AS r
WHERE 500000.00 <= (SELECT AVG(sales_price)
                    FROM sales AS s
```

```
WHERE s.realtor_ID = r.realtor_ID)
```

在本例中，DBMS 必须反复执行相关子查询——当 DBMS 扫描 REALTORS 表时，对每位房地产经纪人都要执行一次。DBMS 每执行一次，相关子查询返回单个（当然是不同的）值（作为未加修饰的比较运算符中的合法判式，子查询必须返回一个标量值）。

技巧 426 使用视图允许子查询中的工作表自我联合

通常，多表查询涉及两个或多个表的关系。例如，假定想要列出那些购买了红色 Corvette 汽车的所有顾客的清单。可以写出将 CUSTOMERS 表内的个人信息与 SALES 表内的购买数据关联起来的如下查询：

```
SELECT first_name, last_name, date_sold, price
FROM customers AS c, sales AS s
WHERE c.cust_ID = s.sold_to
AND make='Corvette'
AND color='red'
```

然而，有时想要编写出涉及自身关系的多表查询。例如，假定想要列出那些在公司工作时间比他们的经理还长的所有雇员的清单。在执行查询时，DBMS 必须把来自表内一行的列值与同一表内的另一行的列值比较。在技巧 237 “在单表 JOIN（即自我 JOIN）中使用表别名”中，读者已学过如何编写涉及单一表内关系的多表查询。在本例中，可以使用以下 SELECT 语句：

```
SELECT e.emp_ID, e.first_name, e.last_name,
       e.date_hired AS 'Employee Hired',
       m.emp_ID AS 'Manager ID', m.date_hired AS 'Manager Hired'
FROM employees AS e, employees AS m
WHERE e.manager_ID = m.emp_ID
AND e.date_hired < m.date_hired
```

通过给 EMPLOYEES 表指定关联名（e 和 m），使得 DBMS 执行自我联合成为可能。在自我联合中，DBMS 处理表内的行，就好像处理两个不同的表（在本例中是表 e 和表 m）中的行一样。DBMS 将来自 EMPLOYEES 表“e”中的行与来自 EMPLOYEES 表“m”中的行联合起来，并返回那些满足查询的 WHERE 子句中的搜索条件的（虚拟）联合行中的列值。

遗憾的是，并不是所有的 SQL 语句都允许对目标表使用关联名。例如，假定将所有雇员的工资减少 25%。他或她受雇时间比经理还早。虽然看起来，为了表达（使用 SQL 术语）想要执行的更新，下面的 UPDATE 语句在句法上是不正确的，因为 UPDATE 语句不允许像下面这样在 UPDATE 子句中指定关联名：

```
UPDATE employees AS m SET salary = salary * 0.75
WHERE employees.manager_ID = m.emp_ID
AND e.date_hired < m.date_hired
```

为了消除上面 UPDATE 语句中非法的关联名 m，可在 EMPLOYEES 表上创建列出所有经理的视图。然后，使用视图作为在 UPDATE 语句的 WHERE 子句中子查询的第二个表。

例如，假定所有非管理雇员在 MANAGER_ID 列里有非 NULL 值，而经理没有。那么以下视图将返回所有经理的清单：

```
CREATE vw_managers
```

```
AS SELECT * FROM employees WHERE manager_ID IS NULL
```

给定返回管理雇员的 DATE_HIRED 和 EMP_ID 列的 VW MANAGERS 视图，可把前述的 UPDATE 语句改写为：

```
UPDATE employees SET salary = salary * 0.75
WHERE EXISTS (SELECT *
              FROM vw_managers AS m
              WHERE employees.manager_ID = m.emp_ID
              AND employees.date_hired < m.date_hired)
```

注意：某些 DBMS 产品（如 MS-SQL Serve）允许引用子查询中的 UPDATE（或 DELETE）语句的目标表，而不用把该表列在子查询的 FROM 子句中。由此，在这些 DBMS 平台上，不必创建视图就可对于查询内的目标表进行外部引用。例如，在 MS-SQL Server 上，可将前面的 UPDATE 语句改写为不使用视图的形式：

```
UPDATE employees SET salary = salary * 0.75
WHERE EXISTS (SELECT *
              FROM employees AS m
              WHERE employees.manager_ID = m.manager_ID
              AND employees.date_hired < m.date_hired)
```

在与子查询的 WHERE 子句中的列值（来自同一表）作比较时，MS-SQL Server 将正确地使用来自对语句的 UPDATE 子句中的 EMPLOYEES 表列的外引用的值。

技巧 427 使用临时表删除重复数据

如果创建没有 PRIMARY KEY 或至少一列受 UNIQUE 和 NOT NULL 约束的表，用户可（而且根据 Murphy 定律，很可能会）插入重复的数据行。表内的重复关系（也就是重复的行）如果不是特意要求的，则很少见。重复的数据不仅要多占空间并增加提取时间和内存用量，而且还防止对数据库的正规化操作。为了防止由于 DELETE、INSERT 和 UPDATE 异常等造成数据库损坏，必须对数据库进行正规化。

正规化过程的第一步是把所有数据库表变成第一正规形式（1NF）。由于第一正规形式的要求之一是表内没有重复的行，消除重复数据是走向所需的没有异常的第三正规形式（3NF）的首要任务之一。

遗憾的是，删除重复数据并不像执行带有 WHERE 子句的 DELETE 语句那样简单。请记住，在执行 DELETE 语句时，将会删除所有满足 WHERE 子句搜索条件的行。这样一来，删除重复数据一行的 DELETE 语句将删除所有重复的行——而没有在表中留下所需要的一行（现在是非重复的）。

从每个重复行组中删除所有行但留下一行的一种方法是执行以下步骤：

1. 使用与含有重复数据的表的一样的结构创建一个临时表。
2. 使用一条带有 DISTINCT 子句的 SELECT 语句的 INSERT 语句把含有重复数据的表行移动到新的临时表中。
3. 截去（也就是删除表内所有的行）原始表。
4. 使用 ALTER TABLE 语句增加一条 UNIQUE 约束，这条约束将防止将来插入重复的行。
5. 使用带有 SELECT 语句的一条 INSERT 语句把临时表中的所有行复制回（现在为）空的原始表中。

6. 删除临时表。

例如，假定有一个下列结构的表名为 ITEMS 的表：

```
ITEM_NUMBER INTEGER
DESCRIPTION VARCHAR(45)
COST MONEY
```

为了从 ITEMS 表中删除重复的行，可执行以下语句批处理：

```
/* Create a temporary table */
CREATE TABLE #temp_items
(item_number INTEGER,
description VARCHAR(45),
cost MONEY)

/* Use the DISTINCT clause to prevent the INSERT statement
   from putting duplicate data into the temporary table */

INSERT INTO #temp_items
SELECT DISTINCT * FROM items

/* Delete all rows from the ITEMS table */

TRUNCATE TABLE items

/* (Optional) - Apply a uniqueness constraint to the
   original table to prevent future duplicates /

ALTER TABLE items ADD UNIQUE (item_number)

/* Return data from the temporary data */

INSERT INTO items
SELECT * FROM #temp_items

/* Drop the temporary table */
DROP #temp_items
```

注意：如果带有重复数据的表具有带 ON DELETE CASCADE 规则的 FOREIGN KEY 约束，或者，如果表有 DELETE 或 INSERT 触发器，请在删除重复行过程之前，确保禁用这些约束和触发器。在把数据从临时表移回到原始表之后，再启用约束和触发器。

技巧 428 使用临时表从多表中删除行

如果 DBMS 既不支持 DELETE 触发器也不支持 FOREIGN KEY 约束的 ON DELETE CASCADE 规则，仍可从多个表中删除行。只要创建一个临时表，其中包括想要删除的行的列表，然后再执行必要的 DELETE 语句从子表，然后从父表中删除相应行。还可以用下面的方法，从还没有用的 FOREIGN KEY 引用建立明确的父/子关系的表中删除“子”行。

为了从没有引用完整性（或 DELETE 触发器）的多个表中删除行，可执行以下步骤：

1. 建立一个临时表，在该表中保存着可用来识别想要在其他表中删除的“子”行的键值（通常是 PRIMARY KEY）。
2. 用一条 SELECT 语句向临时表中插入键值。
3. 用一条在 WHERE 子句中带有子查询的 DELETE 语句从每个相关的表中删除子行。
4. 用一条在 WHERE 子句中带有子查询的 DELETE 语句从由临时表中的键值所确定的父表中删除行。
5. 删除临时表。

例如，假定公司在向顾客寄送预订产品时附送的免费礼品。如果公司随后决定不再继续寄送免费礼品，必须将顾客从 ITEM_MASTER、INVENTORY 和 ORDERS 表中加以删除。为达此目的，可执行以下语句批处理，以便作为单一事务处理从 3 个表中删除行：

```
/* Create a temporary table */
CREATE TABLE #temp_discontinued
(item_number INTEGER)

/* Use the DISTINCT clause to prevent the INSERT statement
   from putting duplicate data into the temporary table */
INSERT INTO #temp_discontinued
SELECT DISTINCT item_number
FROM item_master WHERE type = 'gift'

/* Delete child rows from related tables */

DELETE FROM inventory
WHERE item_no IN (SELECT item_number
                  FROM #temp_discontinued)

DELETE FROM orders
WHERE item_no IN (SELECT item_number
                  FROM #temp_discontinued)

/* Delete the "parent" row from the parent table */

DELETE FROM item_master
WHERE item_number IN (SELECT item_number
                      FROM #temp_discontinued)

/* Drop the temporary table */

DROP #temp_discontinued
```

虽然多次执行同一子查询看起来效率不高，但请切记，DBMS 优化器审阅语句批处理中的所有语句，并在执行之前建立执行计划。当优化器看到反复使用同一子查询时，优化器只执行一次子查询，然后每当子查询出现在语句批处理中时，就使用子查询的（虚拟）结果表（不再执行该子查询）。

注意：虽然在语句批处理末尾可执行以下 DELETE 语句，以便从 ITEM_MASTER 表中删除礼品项，但是最好还是用于查询，如下例所示：

```
DELETE FROM item_master WHERE type = 'gift'
```

将同样的静态子查询结果表用于所有 DELETE 语句，可避免从还没有子表中删除礼品项时就从 ITEM_MASTER 表中删除礼品项的可能性。例如，如果在 DBMS 执行语句批处理内的 INSERT 语句以创建 #TEMP_DISCONTINUED 表之后，一个用户把一件额外的项目标记为礼品，则执行前述的 DELETE 语句可能导致 DBMS 从 ITEM_MASTER 表中删除还没有从其子表中删除的项目。

技巧 429 使用 UPDATE 语句根据另一个表中的值设置表中的值

除非执行维护操作或者改正表内不正确的数据值，通常人们使用数据输入程序接受来自数据库用户的新数据或改变现存的数据。结果，编写的大多数 UPDATE 语句是将列值设置为用户输入到数据输入程序的数据域中的数字或字符串的简单表达式。如果用户输入了一个数字，可根据涉及列的当前值和用户输入的数字的数学公式来设定列值。这样，绝大多数 UPDATE 语句将为以下形式：

```
UPDATE <table name>  
SET <column name> = <variable name or simple expression>  
WHERE <search condition>
```

然而，不要拘泥于本例中 UPDATE 语句的简单形式。例如，没有规则规定不让把列名放在等号的两边。虽然以下 UPDATE 语句并不实际改变列值，但可将其用来触发引用完整性操作。如果正在“更新的”列是被有 ON UPDATE 规则的 FOREIGN KEY 所引用的 PRIMARY KEY，则这样的事就可能发生。如果该列内或者表内有 UPDATE 触发器，还可将其用于执行触发器：

```
UPDATE customers SET cust_no = cust_no
```

此外，不要局限于在 UPDATE 语句的 SET 子句中的等号 (=) 右边只使用列名、变量名、实际值或简单表达式。还可将列值设置为由标量子查询（也就是返回单个标量值的子查询）返回的值。另外，UPDATE 语句的 WHERE 子句还可包含一个子查询，只要该子查询是返回 TRUE 或 FALSE 结果的表达式的一部分。

例如，可使用 SET 子句（在 WHERE 子句之内）里的子查询来编写收集一个表的总计数据并将其放到另一个表列内的 UPDATE 语句。例如，假定有 CUSTOMERS、UNPOSTED_PAYMENTS 和 PAYMENT_HISTORY 表。如要更新每个顾客的 ACCOUNT_BALANCE 和 AVERAGE_PAYMENT 数据（在 CUSTOMERS 表内），可执行以下语句批处理：

```
/* Mark the payments to be posted and insert them into the  
payment history table */  
  
UPDATE unposted_payments SET post = 'Y'  
INSERT INTO payment_history SELECT * FROM unposted_payments  
WHERE post = 'Y'  
  
/* Execute the UPDATE statement that posts the unposted  
payments marked for posting */
```

```

UPDATE customers
SET account_balance = account_balance -
    (SELECT SUM(amount_paid)
     FROM unposted_payments AS up
     WHERE customers.cust_ID = up.cust_ID),

    average_payment =
    (SELECT avg(amount_paid)
     FROM payment_history as ph
     WHERE customers.cust_ID = ph.cust_ID)

WHERE EXISTS (SELECT *
              FROM unposted_payments as up
              WHERE customers.cust_ID = up.cust_ID
                AND up.post = 'Y')

/* Remove posted payments from the unposted payments
   table */

```

```
DELETE FROM unposted_payments WHERE post = 'Y'
```

正如本技巧前边所提到的，用在 UPDATE 语句 SET 子句中的每个 SELECT 语句，必须返回单一标量值。本例使用总计函数确保是一个标量值。虽然有时可能想要把第一个子查询中的 SELECT 子句写作 SELECT amount_paid，但这可能是一个错误。如果一个顾客付款不止一次，SELECT amount_paid 将返回多个值，而整个 UPDATE 进程将失败。

还请注意，UPDATE 语句的 WHERE 子句使用一个子查询来确定顾客是否没有计入支付。如果没有，DBMS 不必计划平均付费，因为 CUSTOMERS 表的 AVERAGE_PAYMENT 列的平均值没有发生任何变化。更为重要的是，当一个客户没有付费，则子查询把空表传给总计函数时，检查是否付费还可确保 AVG() 总计函数不会返回 NULL 值。

在本例中的标量子查询能够从特定顾客处选择付费，因为 UPDATE 子句引用的表对于 UPDATE 语句中所有其余子句都是可用的。因而，在 CUSTOMERS 表内当前行的列值对于整个 UPDATE 语句中的所有子查询都是可用的。

技巧 430 优化 EXISTS 判式

为了检查列值（或实际值）是否处于值的列表中，可使用 IN 判式。例如，如果想要获得那些销售人员不是 Mary、Mark 就是 Sue 的所有顾客的清单，可像下面查询一样使用 IN 判式：

```

SELECT * FROM customers
WHERE salesperson IN ('Mary', 'Mark', 'Sue')

```

不用一个一个地列出值来，可使用一个子查询建立想要检查的可能值的列表。例如，为了从 PROSPECTS 列表中删除所有顾客的电话号码，可写出以下 DELETE 语句：

```

DELETE FROM prospects
WHERE phone_number IN (SELECT phone_number FROM customers)

```

当结果集中至少有一项时，如果想要执行一项操作，可用 EXISTS 判式。例如，在技巧 429 “使用 UPDATE 语句根据另一个表中的值设置表中的值”中，如果在

UNPOSTED_PAYMENTS 表里至少有一个顾客付费, 想让 DBMS 更新客户的账目余额和平均付费数据。如果当顾客支付了特定量的款项, 就只想执行 UPDATE 语句, 则可能写出 UPDATE 语句的 WHERE 子句如下:

```
WHERE customer.normal_payment IN
    (SELECT amount_paid
     FROM unposted_payments AS up
     WHERE customers.cust_ID = up.cust_ID
     AND up.post = 'Y')
```

然而, 由于没有检查特定的 AMOUNT_PAID, 如果 UNPOSTED_PAYMENTS 表包含来自顾客的任意数量的支付, 就想让 DBMS 执行 UPDATE 语句, 则可使用如下 WHERE 子句中的 EXISTS 判式:

```
WHERE EXISTS (SELECT *
              FROM unposted_payments as up
              WHERE customers.cust_ID = up.cust_ID
              AND up.post = 'Y')
```

正如读者将从以下讨论中学习的, 有 3 种方法编写 EXISTS 判式, 而选择哪种方法取决于具体的 DBMS 平台。如要确定对于具体的 DBMS 平台哪一种是最好的 EXISTS 判式, 可尝试用 3 种形式的每一种执行同一查询。可用 DBMS 优化器报告为“代价最低”的那种。当提交一个查询时, DBMS 优化器生成一个执行计划。读者已在技巧 408 “理解显示语句执行计划和状态的 MS-SQL Server SHOWPLAN_ALL 选项”中学过如何查阅一个语句的执行计划。

按照 SQL-89 中的规则, 用在 EXISTS 判式中的子查询必须有包括单列或星号 (*) 的 SELECT 子句。当使用 SELECT * 时, DBMS (至少在理论上) 要从 SELECT 语句的 FROM 子句中的列表挑出一列并加以运用 (请记住, 当使用 EXISTS 判式时, 并不是寻找任何特定类型的特定值, 而只是检查子查询是否返回任何值)。

能处理行值比较的 SQL-92 不再把 EXISTS 判式的子查询局限于列出惟一的单列或星号 (*)。然而, 在本书写作时, 大多数 DBMS 产品仍不支持行值比较。这样, 3 种最常见的 EXISTE 判式如下:

- EXISTS (SELECT * FROM <table name>
 WHERE <search criteria>)
- EXISTS (SELECT <column name> FROM <table name>
 WHERE <search criteria>)
- EXISTS (SELECT <constant value> FROM <table name>
 WHERE <search criteria>)

一般来说, 第一种形式, SELECT * FROM ... 应该以指定列名或者常数值为好。SELECT * FROM ... 允许优化器来决定使用哪一列。因而, 如果用在子查询的 WHERE 子句中的列上有索引, 优化器可使用索引的列, 因而可完全不用触动表的数据。在前面的例子中, 如果在 UNPOSTED_PAYMENTS 表的 CUST_ID 列上有索引, DBMS 只需检查索引 (CUST_ID 列上的)。通常, 索引要比表小, 而且是为快速搜索而构建的。如果 DBMS 在索引内发现值, EXISTS 判式为真; 如果在索引内没发现值, 则判式为假。不管是哪种情况, DBMS 不必从 UNPOSTED_PAYMENTS 表中提取实际的值。

虽然从理论上讲, SELECT * FROM ... 应该是 EXISTS 判式的最佳形式, 但某些 DBMS

产品实际上执行子查询并建立“虚拟”的查询结果表。换言之，不用只检查任何表列之一中的值，DBMS 建立一个内存中的表，其中包含来自那些满足子查询的搜索条件的行中的所有列的值。在这样的系统上，使用 EXISTS 判式的 SELECT <column name>形式把结果表限制为单列，要比让 DBMS 建立一个多列、多行的中间表更为合理。

最后，在诸如 Oracle 一类的 DBMS 产品上，应该使用 EXISTS 判式的第 3 种形式(SELECT <constant value>)，而且必须告诉 DBMS，当建立子查询的虚拟表时，不必提取实际列值。如果提供了实际值（常数），DBMS 只是对每行都重复满足子查询的搜索条件的该值。虽然该种 DBMS 仍不得不扫描子查询的基表，但不必从表列中将实际数据值提取到内存中。

技巧 431 使用 ALL 判式把两个查询合二为一

在技巧 425 “理解对用作比较运算符判式的子查询的限制”中，读者已学过，可用子查询作为比较值之一——只要子查询返回单个标量值。这样一来，可使用标量子查询（也就是返回标量值的子查询）来执行下面的查询，这个查询列出在 EMPLOYEES 表内有最高 TOTAL_SALES 的雇员的姓名和总的销售量：

```
SELECT first_name, last_name, total_sales
FROM employees
WHERE total_sales = (SELECT MAX(total_sales)
                     FROM employees)
```

SQL 提供 3 个量词（有时称为限定词），允许使用在比较中返回值集（多个值）的子查询。这些量词是 ALL、SOME 和 ANY。量词是语句（在此情况下是一个比较）为 TRUE 时确定出对象量的逻辑运算符。这样，可以把前面的查询改写为：

```
SELECT first_name, last_name, total_sales
FROM employees
WHERE total_sales >= ALL (SELECT total_sales
                          FROM employees
                          WHERE total_sales IS NOT NULL)
```

如果在 EMPLOYEES 表中有两行或不止两行带有 TOTAL_SALES 数字，那么，本例中的子查询将要返回多个值。然而，第二个查询将返回与第一个查询一样的结果集。仅当某个特定雇员的 TOTAL_SALES 值大于或等于 ALL 限定词之后的子查询返回的每个 TOTAL_SALES 值时，第二个子查询中的 WHERE 子句才求值为 TRUE。

注意：当用 ALL 限定词时，必须清除 NULL 值。MAX()和 MIN()函数可自动做到这一点。在应用 ALL 限定词时，DBMS 把比较运算符左边的标量值同限定词后的子查询返回的每个值进行比较。为了让 WHERE 子句求值为 TRUE，每个执行的比较必须求值为 TRUE。当把任何值与 NULL 比较时，结果是 UNKNOWN（未知的）。结果，如果子查询的结果集包含任何 NULL 值，WHERE 子句将求值为 FALSE，因为执行的所有比较都没有求值为 TRUE。

在这种情况下，第一个查询比第二种查询更直观，因此比第二个查询更容易理解。事实上，无论何时，如有涉及把一个标量值同可写为标量子查询的子查询的结果进行比较的查询，人们将发现，当使用比较运算符而不是限定词来编写查询时更容易理解。当你必须比较两个累积值时，限定词是便利的工具。

例如，假定想要列出具有最大数目的房屋合同的房产经纪人。如果不用限定词，必须写

出两个查询（一个在视图内），才能得到想要的答案：

```
CREATE VIEW vw_total_listings(realtor_ID, listing_count)
SELECT realtor_ID, COUNT(*)
FROM listings
GROUP BY realtor_ID

SELECT realtor_ID, listing_count
FROM vw_total_listings
WHERE listing_count = (SELECT MAX(listing_count)
                      FROM vw_total_listing)
```

请注意，总结列表的查询在视图内是隐藏的，从而可把累积列表计数作为比较运算符左边的标量值用于每个房产经纪人。通过使用限定词 ALL，就可将同样的查询写作显示总计计算值的单条 SELECT 语句：

```
SELECT realtor_ID, COUNT(*)
FROM listings
GROUP BY realtor_ID
HAVING COUNT(*) >= ALL (SELECT DISTINCT COUNT(*)
                        FROM listings
                        GROUP BY realtor_ID)
```

技巧 432 使用 EXISTS 判式检查表中的重复行

SQL-92 标准包括一个 UNIQUE 判式——不要把它同 UNIQUE 列约束混淆。尽管 UNIQUE 列约束能防止用户在一列中插入重复值，但 UNIQUE 判式却可测试由子查询返回的结果集中有无重复行。如果子查询的结果集不包含任何重复行，则 UNIQUE 判式返回 TRUE，如果包含，则返回 FALSE。这样，为了检验在表中的每一行内的列（或多列）是否包含惟一值，可写出如下形式的查询：

```
SELECT 'All Unique'
WHERE UNIQUE (SELECT <column list> FROM <table name>)
```

当执行以上查询（当然要有合法的列和表名）时，如果由<column list>给出的该列（或者多列中）的值不同于由<table name>指定的表的每行内的值，则本例中的 SELECT 语句显示单词 All Unique。

如果与逻辑连接符 NOT 一起使用 UNIQUE 判式，可检查表中一列（或者多列）重复的值。例如，为了确定 ITEM_MASTER 表中的 ITEM_NUMBER 列是否有重复的值，可在 UNIQUE 判式前加 NOT，并写出如下查询：

```
SELECT DISTINCT 'Contains Duplicates'
WHERE NOT UNIQUE (SELECT item_number FROM item_master)
```

遗憾的是，许多 DBMS 产品都不支持 UNIQUE 判式。然而，可用 EXISTS 判式（在所有商用的 DBMS 系统中都有这项功能）执行同样的“惟一性”测试。例如，可改写前面的查询，该查询检查 ITEM_NUMBER 列内重复的值，如下所示：

```
SELECT DISTINCT 'Contains Duplicates'
WHERE EXISTS (SELECT item_number, COUNT(*)
              FROM item_master
              GROUP BY item_number)
```

```
HAVING COUNT(*) > 1)
```

技巧 433 把表内容和函数结果合并

在处理 SQL 数据一段时间后，人们将会注意到，所编写的大多数查询都把来自两个或多个表的数据结合起来。然而，SQL 并不将人们局限为只能把一个表中的行与另一表中的相关行结合起来。还可以把由函数调用返回的结果与表中的行结合起来。

例如，假定某人负责为一个政党保存调查结果。为了把所需的存储量减到最小，以便存放大量的回应，可把问题和回应分别放到两个表里。在调查答卷表中每行都由一串 1 和 0 组成，1 代表对调查问题的肯定回答，0 代表对调查问题的否定回答。SURVEY_QUESTIONS 和 SURVEY_RESPONSES 表可定义如下：

```
CREATE TABLE survey_questions
(survey_number SMALLINT,
 question_number TINYINT,
 question_text VARCHAR(256))
```

```
CREATE TABLE survey_responses
(survey_number SMALLINT,
 response_string VARCHAR(50))
```

RESPONSE_STRING 所需的字符数量将取决于调查中问题的数量。在本例中，回答字符串可处理多达 50 个回答（也就是有多达 50 个 1 和 0），分别对 50 个问题中的每个问题给出是或否的回答。

对调查的 QUESTION_NUMBER 为 1 的问题的回答保存在 RESPONSE_STRING 列中的字符 1 中。对调查的 QUESTION_NUMBER 为 2 的问题的回答保存在 RESPONSE_STRING 的字符 2 中，以此类推。这样一来，SURVEY_RESPONSES 表中的 RESPONSE_STRING 列的值为 1010 将表示人们对被调查问题 1 和问题 3 的回答为是，而对被调查问题 2 和问题 4 的回答为否。

现在，假定把下面 4 个问题作为 SURVEY_NUMBER 为 1 的问题装入 SURVEY_QUESTIONS 表中：

```
1,1,'Do you support home schooling?'
1,2,'Do you want public school vouchers?'
1,3,'Do you support lower taxes over deficit reduction?'
1,4,'Do you support placing the nuclear repository in Nevada?'
```

并把下面的回答放到 SURVEY_RESPONSES 表：

```
1,'1111'
1,'1000'
1,'1100'
1,'1110'
1,'0000'
```

为了根据存储在 SURVEY_QUESTIONS 和 SURVEY_RESPONSES 两个表里的调查数据生成报告，还需要如下所示的第 3 个表，其中合并了特定调查的问题和回答：

```
CREATE TABLE survey_results
(question VARCHAR(256),
```

```
response CHAR(1))
```

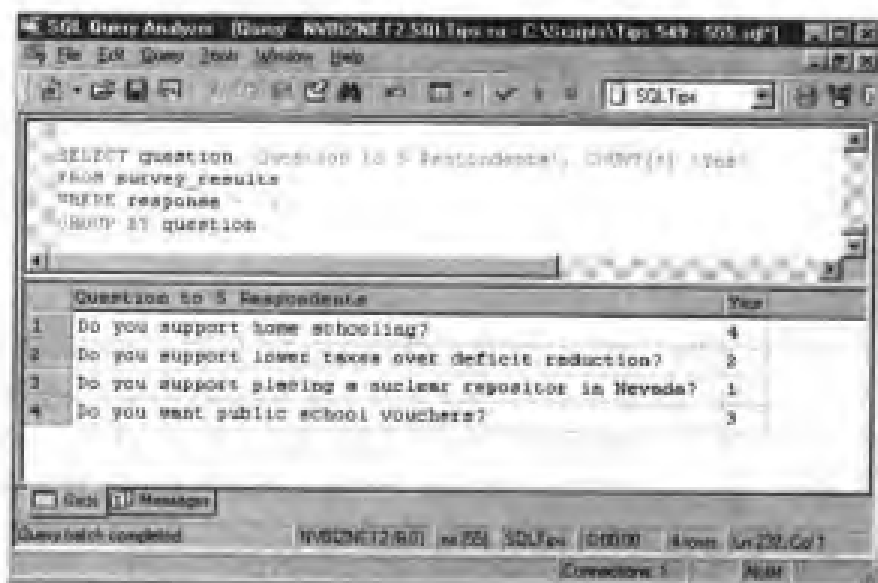
然后，为了把调查“1”的结果放入 SURVEY_RESULTS 表，可用以下 INSERT INTO 语句：

```
INSERT INTO survey_results (question, response)
  SELECT question_text,
         SUBSTRING(response_string, question_number, 1)
  FROM survey_responses, survey_questions
 WHERE survey_responses = 1
       AND survey_questions = 1
```

SELECT 语句将来自 SURVEY_QUESTIONS 的一行与由 SUBSTRING() 函数返回的结果合并。然后 INSERT INTO 语句将“合并”后的行插入 SURVEY_RESULTS 表。当查询从 QUESTION_NUMBER 1 移动到 QUESTION_NUMBER *n* 时，QUESTION_NUMBER 参数将 SUBSTRING() 函数从左到右一次移过 RESPONSE_STRING 的一个字符。这样，不论有 4 个问题还是 50 个问题，都不必修改查询。根据问题的数量，查询自动地调节本身，一次一个回答地处理任意长的 RESPONSE_STRING。

为了根据 SURVEY_RESULTS 表的信息生成报告，可执行一组 SELECT 语句，如下所示，以便产生如图 23.1 所示的调查问题和回答：

```
SELECT question 'Question to 5 Respondents', COUNT(*) 'Yes'
FROM survey_results
WHERE response = '1'
GROUP BY question
```



	Question to 5 Respondents	Yes
1	Do you support home schooling?	4
2	Do you support lower taxes over deficit reduction?	2
3	Do you support placing a nuclear repository in Nevada?	1
4	Do you want public school vouchers?	3

图 23.1 合并了调查问题和回答工作表的组合查询结果

技巧 434 使用视图显示汇总级别的层次

大多数商务都是以分层形式组织起来的，其中，随着命令链向上移动，从一般工作人员向执行总裁的升迁，职责和控制的跨度都在增加。在各层的每个级别上，经理需要总结报告以了解其下属的工作的好坏。例如，在销售组织中，单个销售都组合在产生销售业绩的销售人员之下。然后部门经理可根据总的销售量比较销售人员的销售业绩。接着，销售按部门在地区内

组合在一起,从而地区负责销售的副总裁可获得对他或她的部门经理工作的好坏的感觉。最后,再把各地区的销售汇总而成的总的销售信息提供给公司执行总裁,他或她必须把公司总的销售增长(或者下降)向董事会或股东大会报告。

如要建立这种分层的汇总信息,可用一系列 SELECT 语句和视图累积各层的汇总。在完成各层总结报告时,每个视图都要根据其前一步中一层的汇总情况。例如,可建立以下视图按雇员、部门和地区总结公司的销售数据:

```
CREATE VIEW vw_salesrep_sales (region, department,
                               salesrep_ID, total_sales)
AS SELECT region, department, salesrep_ID SUM(amount)
FROM sales
GROUP BY region, department, salesrep_ID
```

```
CREATE VIEW vw_department_sales (region, department,
                                  total_sales)
AS SELECT region, department, SUM(total_sales)
FROM vw_salesrep_sales
GROUP BY region, department
```

```
CREATE VIEW vw_regional_sales (region, total_sales)
AS SELECT region, SUM(total_sales)
FROM vw_department_sales
GROUP BY region
```

为了生成总结报告,可根据前一视图执行 SELECT 语句,如下所示:

```
SELECT * FROM vw_salesrep_sales
SELECT * FROM vw_department_sales
SELECT * FROM vw_regional_sales
```

```
SELECT SUM (total_sales)
FROM vw_regional_sales
```

当作为语句批处理运行时,执行这些对视图的 SELECT 语句很可能要比直接根据基表(SALES)的4个查询要快。如果查询 SALES 表而不使用视图,DBMS 必须反复(每个查询一次)扫描 SALES 表的所有行。如果使用视图,DBMS 只需扫描 SALES 表一次。其后,DBMS 扫描在物化层次内的前一级的视图时创建的虚拟(既可在内存中也可在磁盘缓存中)表。

技巧 435 使用带标量子查询的 SELECT 语句显示流水总计

虽然 SQL 是一种数据存取的优秀语言,但它并不是报告编写程序。SQL 可容易地计算一系列的总和、平均值、最小值或最大值。然而,SQL 提供一组行内多列的总计的能力有限。尽管一个报告编写程序能多次扫描报告数据,以便在提供详细的行的同时,还提供组(以及由组组成的组)的描述性统计数字,但如果想用 SQL 组总计,则 SQL 的求总计的能力(由 GROUP BY 子句提供)的限制则不得不放弃细节行。

例如,一个报告编写程序可提供每位销售人员的总销售量、所有销售人员的总的销售量以及提供每笔销售信息的数据行。如果想让 SQL 按每位销售人员求总计的话,必须执行如下

的组合查询，并放弃详细的销售信息：

```
SELECT salesrep_ID, SUM(amount_invoiced) 'Total Sales'
FROM sales
GROUP BY salesrep_ID
ORDER BY salesrep_ID
```

虽然通过使用主查询的 SELECT 子句中的子查询（如下面所示）已经既报告了总量，也报告了细节，但在报告的一列往下重复相同的总销售数字，却令人混淆：

```
SELECT *, (SELECT SUM(amount_invoiced)
           FROM sales AS s2
           WHERE s2.salesrep_ID = s1.salesrep_ID) AS
           'Total Sales'
FROM sales AS s1
ORDER BY salesrep_ID, invoice_date
```

尽管在每个细节行上都显示总销售量并不是所希望的，但用一个标量子查询提供流水总计，有时能提高报告的可用性。例如，假定有一个有支票（或者提款）和存款交易的表。如果 TRANS_AMOUNT 列中提款为负值，而存款部分为正值，可使用以下查询列出交易记录以及存款小于提款的流水总计：

```
SELECT t1.trans_date, trans_type, trans_amount AS 'Amount'
      (SELECT SUM(t2.trans_amount)
       FROM transactions AS t2
       WHERE t2.trans_date <= t1.trans_date) AS
       'Net (Deposits - Checks)'
FROM transactions AS t1
ORDER BY trans_date
```

当然，为了真正有用，最好在存储过程或语句批处理中执行前述查询，这样就可在诸如变量 @ACCT_NO 内放置账户计数器，而在变量 @BEGIN_BAL 内放置账户的开始余额。然后可把开始余额加到净交易量上以显示流水账户余额，并把查询限制为只显示那些特定账户的交易记录（而不是 TRANSACTIONS 表内的所有支票和存款）：

```
SELECT t1.trans_date, trans_type, trans_amount AS 'Amount'
      (SELECT SUM(t2.trans_amount)
       FROM transactions AS t2
       WHERE t2.trans_date <= t1.trans_date
       AND t2.acct_number = @acct_no) + @begin_bal)
      AS 'Account Balance'
FROM transactions AS t1
WHERE t1.acct_number = @acct_no
ORDER BY trans_date
```

技巧 436 使用 EXCEPT 判式确定两表差异

集合差运算符 EXCEPT 允许确定 TABLE_A 中的在 TABLE_B 中找不到的行。这样一来，如果在表 TABLE_A 中装入所有学生的清单而把那些完成了必修课的学生信息装入表 TABLE_B，则可使用以下语句显示那些除完成必修课的学生以外的所有学生：

```
SELECT * FROM table_a
```

```
EXCEPT
```

```
SELECT * FROM table_b
```

请注意，EXCEPT 运算符在把第一个查询（SELECT * FROM TABLE_A）与第二个查询（SELECT * FROM TABLE_B）比较之前，已从第一个查询的结果集中清除了所有重复信息。这样一来，从 EXCEPT 查询中得到的结果集中绝不会包含任何重复行。

虽然本例中的查询假定 TABLE_A 和 TABLE_B 是与 UNION 运算兼容的，但 EXCEPT 语句并不要求 TABLE_A 和 TABLE_B 是与 UNION 运算兼容的。正如读者以前所学过的，如果两个表的列数相同，而且每个表中每行的数据类型与另一个表中相对应列（顺序位置）的数据类型也相同，那么这两个表是与 UNION 运算兼容的。

由于 EXCEPT 运算符返回那些第一个结果集中的没有出现在第二个结果集中的所有行，因而通过简单地从每个表中列出运算符将要比较的列，可把 EXCEPT 运算符用于两个并不类似的（也就是说非 UNION 运算兼容的）表。这样一来，为了列出所有不是雇员的顾客，可以写出下列查询：

```
SELECT first_name, last_name, phone_number FROM customers
EXCEPT
SELECT first_name, last_name, phone_number FROM employees
```

遗憾的是，许多 DBMS 产品还没有实现 EXCEPT 运算符，因而，可能不得不使用 LEFT OUTER JOIN 运算符。例如，假定购买了一张有前景的顾客清单，在把此清单交给销售人员之前，想要先清除当前已有的顾客，可把有前景的清单数据装入名为 PROSPECTS 的表中，并使用以下 LEFT OUTER JOIN 生成那些不在 CUSTOMERS 表中的 PROSPECTS 清单：

```
SELECT DISTINCT prospects.*
FROM (prospects LEFT OUTER JOIN customers
ON prospects.phone_number = customers.phone_number)
WHERE customers.phone_number IS NULL
```

前面的查询假定 PHONE_NUMBER 既可是 PRIMARY KEY 也可是其值惟一确定单独的潜在顾客和当前顾客的列。

注意：某些 DBMS 产品以不同的名称实现 EXCEPT 运算符。例如，Oracle 使用 MINUS。因此，在用效率较低的 LEFT OUTER JOIN 替换 EXCEPT 时，请检查一下 DBMS 文档，看看是否实现了 EXCEPT。

技巧 437 使用 EXISTS 判式生成两表的交集

当想要知道两个表里那些行是共用的，可使用 INTERSECTION 运算符。虽然 EXCEPT 运算符（读者已在技巧 436 “使用 EXCEPT 判式确定两表差异”中已学过有关知识）返回那些在第一个结果集中而不在第二个结果集中的行，但 INTERSECT 运算符只返回那些在两个结果集中都出现的行。这样，如果贵公司销售保险，而且想要得到从贵公司既购买了人寿险也购买了车险的顾客清单，可使用类似于以下的查询：

```
SELECT cust_ID, first_name, last_name, phone_number
FROM life_cust_list
INTERSECT
SELECT cust_ID, first_name, last_name, phone_number
FROM auto_cust_list
```


如同 EXCEPT 运算符一样，在 INTERSECT 运算符中用的两个表不必是 union 兼容的。然而，在 INTERSECT 关键词的前后必须返回同样数量的列，而且相对应的列必须是兼容的数据类型。除了只返回在两个结果集中都出现的行之外，INTERSECT 运算符还清除所有的重复行。

现在，只有少数 DBMS 产品实现了 INTERSECT 运算符。然而，可使用带 EXISTS 判式的 SELECT 语句产生两个表的交集。例如，如果想要生成既在教务长清单（Dean List）又在橄榄球队中出现的学生清单，可以生成以下查询：

```
SELECT DISTINCT student_id, first_name, last_name, GPA
FROM students
WHERE EXISTS (SELECT * FROM football_team
              WHERE students.student_ID =
                football_team.student_ID)
```

还可用 INNER JOIN 写出同样的查询：

```
SELECT DISTINCT student_id, first_name, last_name, GPA
FROM students INNER JOIN football_team
ON students.student_ID = football_team.student_ID
```

第 24 章 探索 MS-SQL Server 的内建存储过程

技巧 438 使用 sp_detach_db 和 sp_attach_db 在 MS-SQL Server 上删除和添加数据库

为了从 MS-SQL Server 上删除数据库——同时把该数据库的数据 (.mdf) 和事务处理日志 (.ldf) 文件原封不动地留在硬盘上——可调用内建的存储过程 sp_detach_db。虽然没有与 MS-SQL Server 连接时，人们不能访问数据库内的数据，但可把数据库的 .mdf 文件（其中包含数据库数据及其所有对象）完全复制或移动到另一个硬盘驱动器或者另一台服务器上。然后，正如我们在本技巧后面要讨论的那样，可重新把数据库文件（和可选的事务处理日志文件）与同一或者另一个不同的 MS-SQL Server 连接。当这项工作完成后，就可再次使用数据库内的数据和对象。

内建的存储过程 sp_detach_db 允许从 MS-SQL Server 中删除数据库而不从硬盘上删除数据库文件，其句法如下：

```
sp_detach_db [@dbname=] '<database name>'
[, @skipchecks=] { 'TRUE' | 'FALSE' }
```

其中：

- @dbname 是想要从 MS-SQL Server 中删除的数据库名。
- @skipchecks 指定在从 DBMS 分开数据库之前，MS-SQL Server 是否可对数据库内的每个表和索引的视图运行 Transact-SQL UPDATE STATISTICS 语句。如果此参数为 FALSE 或 NULL，DBMS 运行 UPDATE STATISTICS；如果为 TRUE（如果不为 @SKIPCHECKS 指定值，则 TRUE 是默认值），DBMS 不能运行 UPDATE STATISTICS。

请注意，MS-SQL Server 保存着对索引内键值分配的统计，还保存着（有时）在某些表的非索引列内存储的值的相同的统计。优化器使用这些统计来确定在执行查询时，使用哪个索引或表列。无论何时存储在表内的值有很大变化，或者如果某人添加了大量的数据（使用块 INSERT INTO 语句）或者删除了大量的数据（使用 TRUNCATE 语句），都应把 @SKIPCHECKS 设定为 FALSE，从而 DBMS 将更新表和索引统计。此外，如果计划把 DBMS 移到只读装置上，可把 @SKIPCHECKS 设定为 FALSE，从而使得永久性数据库有最新的索引，这将允许查询尽可能有效地和尽可能快地提取数据。否则，可把 @SKIPCHECKS 设定为 TRUE 或者在存储过程调用中忽略其值。

如要把数据库 SQLTips 从 MS-SQL Server 上删除，而不更新有关 INDEXES 和表列值的统计，可向 DBMS 提交以下 EXEC 语句：

```
EXEC sp_detach_db @dbname='SQLTips', @skipchecks='TRUE'
```

当必须重新把数据库 (.mdf) 文件与以前使用 sp_detach_db 从其上断开连接的 MS-SQL Server 连接时，可用内置的存储过程调用 sp_attach_db，其句法如下所示：

```
sp_attach_db [@dbname=] '<database name>'
    ,[@filename<n>=]
    '<pathname of an .mdf or .ldf file>'
    [...,@filename16]
```

其中:

- @dbname 是想要连接到 MS-SQL Server 上的数据库名。不必使用与以前连接 MS-SQL Server 相同的数据库名, 用任何一个有效的数据库名都可以。
- @filename<n> 是数据库 (.mdf) 或事务处理日志 (.ldf) 文件的全路径名。虽然 .ldf 文件保存着数据库事务处理日志, 但 .mdf 文件保存着所有数据库的数据、对象、用户以及角色信息。

例如, 假定想要重新连接 SQLTips 数据库, 其数据储存在文件: C:\MSSQL\Data\SQLTips_data.mdf 内, 事务处理日志储存在文件: D:\MSSQL\LogFiles\SQLTips_log.ldf 内。为了重新把 SQLTips 与 MS-SQL Server 连接, 可执行下列 EXEC 语句:

```
EXEC sp_attach_db @dbname='MySQLTips',
    @filename1='C:\MSSQL\Data\SqlTips_data.mdf',
    @filename2='D:\MSSQL\LogFiles\SQLTips_log.ldf'
```

如果只有数据库.mdf (数据) 文件, 则可如下调用 sp_attach_db:

```
EXEC sp_attach_db @dbname='MySQLTips',
    @filename1='C:\MSSQL\Data\SqlTips_data.mdf'
```

当只有数据库.mdf (数据) 文件时, 存储过程将把数据文件连接到 MS-SQL Server 上, 然后创建新的事务处理日志。

当把数据库重新与 MS-SQL Server 连接时, 则以前曾经被授予对此数据库访问权的用户又可以访问该数据库了。所有数据库对象和数据以及用户和角色定义都储存在数据库.mdf 文件内。因而, 执行 sp_attach_db 过程后, 在断开连接以前在原来数据库内所创建的对象、数据、登录以及安全性, 在重新与 MS-SQL Server 连接之后都是可用的。

技巧 439 使用 MS-SQL Server 的存储过程 sp_addtype 和 sp_droptype 添加和删除用户定义的数据类型

在创建表时, 你必须为每列指定数据类型, 从而定义该列可保存数据的类型。例如, 如果一列的类型为 INTEGER, 用户和应用程序只能在该列存储整数——字符和带小数点的数字是不允许的。类似地, 当把一列定义为 CHAR(10) 类型时, 该列只能储存 10 个字符、符号或者数字。

通过创建用户定义的数据, 可为标准的 SQL 数据类型指定一个描述性的名称。所指定的名称应该为用户描述他们在列内发现的数据类型和/或数据值的范围。例如, 假定正在处理 EMPLOYEE 表中 SALARY 列。可把该列的数据类型定义为 NUMERIC(10,2), 或者使用更具描述性的用户定义的数据类型, 如 EXECUTIVE_SALARY、MANAGER_SALARY、或者 SUPERVISOR_SALARY。另外, 通过创建一条规则并把该规则绑定于指定到列的用户定义的数据类型, 则可保证输入到列 (在本例中是 SALARY), 落在某一值范围内。

技巧 36 “使用 MS-SQL Server Enterprise Manager 创建用户定义的数据类型” 已经向读者展示了如何创建用户定义的数据类型。除了在 Enterprise Manager 中创建数据类型之外,

MS-SQL Server 还允许在命令行上调用内建的存储过程 `sp_addtype` 创建数据类型, 其句法如下:

```
sp_addtype [@typename=] '<data type name>'
           [,[@phystype=] '<valid system data type>'
           [,[@nulltype=] '{NULL|NOT NULL|NONNULL}'
           [,[@owner=] '<username>']
```

其中:

- `@typename` 是描述表列内可保存的数据类型或者数值范围的名称。例如, 可用像 `HOURLY_PAYRATE`、`EXECUTIVE_SALARY` 和 `SSAN` 这样的名称作为用户定义的数据类型。
- `@phystype` 是 DBMS 产品上合法的内建的数据类型。绝大多数 DBMS 产品支持 SQL 规范内定义的所有数据类型, 而且还另外添加少量的其自己的数据类型。因此, 请检查一下系统文档, 以便找出可指定给 `@PHYSTYPE` 参数的预先定义的数据类型的完全列表。
- `@nulltype` 指定将要定义为用户定义数据类型是否可保存 `NULL` 值。当在 `CREATE TABLE` 或者 `ALTER TABLE` 语句中使用用户定义的数据类型时, 通过提供不同的设置, 可以推翻这个默认的可保存 `NULL` 值的设置。
- `@owner` 指定拥有正要创建的数据类型的用户名。如果没有通过 `@OWNER` 参数将用户名传递到存储过程, 执行 `sp_addtype` 存储过程的用户将成为新的数据类型的所有者。

例如, 为了创建用户定义的数据类型 `SALES_TAX`, 其中定义最多 6 位数字的数值, 小数点后面有 5 位数字, 可如下调用 `sp_addtype`:

```
EXEC sp_addtype @typename='SALES_TAX',
               @phystype='NUMERIC(6,5)', @owner='dbo'
```

请注意, 本例中的 `sp_addtype` 存储过程调用使得 `DBO` 成为用户定义的数据类型的所有者。无论何时使 `DBO` 成为用户定义的数据类型的所有者, 所有用户可通过名称引用新的数据类型。如果一位不是 `DBO` 的用户拥有某一数据类型, 在列定义内使用该数据类型时, 必须既提供所有者的用户名也提供用户定义的数据类型名。

当不再需要某一数据类型时, 可使用以下句法调用 `sp_droptype` 存储过程将其从数据库里删除:

```
sp_droptype [@typename=] '<data type name>'
```

其中:

- `@typename` 是想要删除的用户定义的数据类型名。

注意: 只能丢弃 (也就是删除) 当前未使用的用户定义的数据类型。因而, 必须先把应用于列的该数据类型从所有表定义中删除, 然后才能删除用户定义的数据类型。此外还必须撤销所有以前与该数据类型绑定的规则和默认值 (使用 `sp_unbindrule` 或者 `sp_unbindefault`)。

这样一来, 为了从 DBMS 中删除用户定义的类型 `SSAN`, 可用以下语句:

```
EXEC sp_droptype @typename='SSAN'
```

技巧 440 使用 `sp_help` 显示数据库对象属性

内建的存储过程 `sp_help` 可显示数据库内对象的属性。正如在 Windows 中, 当右击对象并弹出菜单中选择 `Properties` 命令后显示对象描述一样, `sp_help` 返回作为参数传递到存储过程

中的对象的描述。

sp_help 存储过程调用的句法为：

```
sp_help [[@objname=]<database object name>]
```

其中：

- @objname 是想让 DBMS 描述属性的数据库对象名。@OBJNAME 可以是任何数据库对象，包括用户定义的数据类型，或者表、索引、约束、视图、存储过程等名称。

如果调用 sp_help 而没有提供一个对象名（在 @OBJNAME 参数内），如以下代码行所示，MS-SQL Server 将返回列出数据库内每个对象名称、所有者和数据类型的结果集：

```
EXEC sp_help
```

调用 sp_help 而不提供对象名是一种方便的获得所有数据库对象列表的方法。然后，可一次传递给 sp_help 一个对象，从而获得该对象的额外信息。

调用 sp_help 时 DBMS 返回的结果集（也就是具体属性信息）取决于通过 @OBJNAME 参数传递到存储过程的对象类型。例如，如果传递了一个约束名，DBMS 将返回一个表，其中列出以下信息：

- CONSTRAINT_TYPE——约束类型。
- CONSTRAINT_NAME——约束名。
- DELETE_ACTION——对于 FOREIGN KEY 约束来说，既可是 Cascade，也可是 No Action。并不适用于所有其他约束。仅当 FOREIGN KEY 定义有 ON DELETE CASCADE 选项时，DELETE_ACTION 才是 Cascade。
- UPDATE_ACTION——对于 FOREIGN KEY 约束来说，不是 Cascade 就是 No Action。不适用于所有其他约束。并不适用于所有其他约束。仅当 FOREIGN KEY 定义有 ON UPDATE CASCADE 选项时，UPDATE_ACTION 才是 Cascade。
- STATUS_ENABLED——表明不论是 FOREIGN KEY 还是 CHECK 约束，都是启用的。不适用于所有其他约束类型。
- STATUS_FOR_REPLICATION——表明不论是 FOREIGN KEY 还是 CHECK 约束，在复制时都要实施，不适用于所有其他约束类型。
- CONSTRAINT_KEYS——组成约束的列名，或者对于默认值、规则以及 CHECK 约束来说，是定义约束的文本。

类似地，如要显示有关存储过程如 USP_PROCESS_CHECK 的信息，可如下调用 sp_help：

```
EXEC sp_help 'USP_PROCESS_CHECK'
```

MS-SQL Server 将显示存储过程名、其所有者的用户名以及创建的日期和时间。接下来，DBMS 将返回带有有关每个存储过程参数信息的结果集。结果集包括：

- PARAMETER_NAME——参数变量名，如 @account_number、@check_number 和 @check_date 等。
- TYPE——参数的数据类型。
- LENGTH——参数的最大物理储存尺寸（单位为字节）。
- PREC——总位数（对于数字参数来说）或者字符数（对于非数字参数来说）。
- SCALE——对于数字参数来说，小数点右边允许的位数，其他情况下为 NULL。
- PARAM_ORDER——参数的顺序位置，也就是 1 代表第 1 个参数，2 代表第 2 个参

数，3 代表第 3 个参数，依此类推。

技巧 441 使用 sp_helptext 显示定义存储过程、用户定义函数、触发器、默认值、规则或者视图的文本

MS-SQL Server 在 SYSCOMMENTS 表行的 TEXT 列中保存着在定义存储过程、用户定义函数、触发器、默认值、规则或者视图时所输入的语句批处理。如果在创建这些对象时，没有对存储过程、函数或触发器加密的话，可用以下句法，通过调用存储过程 sp_helptext 来显示其语句：

```
sp_helptext [@objname=<database object name>]
```

其中：

- @objname 是存储过程、用户定义数据类型、函数、触发器、默认值、规则或者视图名。

只能用 sp_helptext 来显示当前数据库内的对象的语句批处理。例如，内建的存储过程 sp_helptext 是在 MASTER 数据库内定义的。由此，为了显示存储过程 sp_helptext 的定义，必须首先执行 USE master 语句，然后再调用存储过程 sp_helptext，如下所示：

```
EXEC sp_helptext 'sp_helptext'
```

如果在创建对象的 CREATE 语句中包括 WITH ENCRYPTION 子句对对象定义加了密，sp_helptext 将不能显示对象的文本。此时 sp_helptext 将显示信息 “The object <encrypted object name> has been encrypted.”（对象<加密的对象名>已被加密）。当然，Sp_helptext 将以对象名代替信息中的<encrypted object name>部分。

技巧 442 用 sp_depends 显示定义视图的表和（或）视图

在改变表或视图之前，特别是当改变不管哪种对象类型的列的数量或顺序之前，一定要调用 sp_depends。正如读者在技巧 8 “理解视图” 中所学过的，视图是虚拟表，其列和行的数据是从数据库内的其他视图或基表中派生出来的。因此，如果要删除一个表，那么引用了该表内列值的视图就不再好用了。当用户查询一个底层表已经删除的视图时，DBMS 返回如下出错信息：

```
Server: Msg 208, State 1, Procedure vw_show_high_rollers,  
Line2
```

```
Invalid object name 'high_rollers'.
```

```
Server: Msg 4413, Level 16, State 1, Line 1
```

```
Could not use view or function 'vw_show_high_rollers'  
because of binding errors
```

虽然实际出错信息的文本可能有所不同，但其要点是一样的：基表（或基视图）已经删除的视图，不能显示数据。此外，由于向存储过程、用户定义的函数反馈数据的视图停止工作而使存储过程以及用户定义的函数停止工作，则可能导致整个 DBMS 的级联错误。为了防止数据链中断，必须在删除其他视图赖以存在的表或视图之前，删除其数据依赖性。

当数据库对象 A 引用数据库对象 B 内的一列时，就说对象 A 依赖于对象 B。例如，当视图的一列引用了表内的列时，视图依赖于表。类似地，当一个视图的列之一引用了另一视图的一列时，则前一视图依赖于后一个视图。

MS-SQL Server 在 SYSDEPENDS 表内标记了数据库对象间的所有依赖性。遗憾的是，SYSDEPENDS 是通过 ID 号而不是名称来引用所有数据库对象的。结果，通过查询 SYSDEPENDS 来检查哪个对象是有依赖性的，是件具有挑战性的事情。幸运的是，可使用内建的存储过程 sp_depends 通过名称来检查对象之间的依赖性。

sp_depends 存储过程调用的句法如下：

```
sp_depends ([@objname=]<database object name>]
```

其中：

@objname 是想要检查依赖性的视图、表或者其他数据库对象名。

sp_depends 不仅报告那些依赖于传递到存储过程中的对象名（通过 @OBJNAME 参数）的对象，而且还要报告对象本身所要依赖的数据库对象。这样一来，例如在删除或改变视图 VW_HIGH_ROLLERS 之前，调用 sp_depends（如以下代码所示）以确定是否有任何数据库视图引用了 VW_HIGH_ROLLERS 视图内的列：

```
sp_depends 'vw_high_rollers'
```

如果 sp_depends 报告有数据库对象依赖于想要删除的视图，则不要删除此视图。或者，如果硬要删除该视图，那么既可改变依赖对象的引用，从而使之引用具有相同数据的其他数据库对象，也可以删除依赖对象视图，因为这些视图不再工作了。当然，在删除任何依赖对象之前，应该使用 sp_depends 来检查一下是否有其他对象依赖于要删除的对象。

技巧 443 使用 sp_helpconstraint 显示有关表约束的信息

正如读者在技巧 11 “理解约束”中所学过的，约束是一种限制用户或应用程序可放入表列的值范围或数据类型的数据对象。有 7 种类型的约束：断言、域、检查约束、外键约束、主要键约束、所需数据和唯一性约束。技巧 11 解释了每种类型约束在维持数据库完整性方面所起的作用。底线是 DBMS 防止用户和应用程序插入任何对表列或对整个表违反约束的数据行。

如果正在手工插入行（例如，通过 SQL Query Analyzer 执行 INSERT 语句），当它拒绝正在试图插入的行时，MS-SQL Server 将在屏幕上报告违反约束的情况。当使用外部应用程序向数据库表插入数据或者当要求存储过程这样做时，DBMS 仍拒绝插入有非法值的行。然而，如果应用程序或者存储过程不能恰当地处理错误，可能永远都看不到错误信息——用户可能认为已被插入数据库的数据实际上已经丢失。

当编写插入或更新表数据的批处理程序时，理解表列的约束是重要的。因此，在编写存储过程或者更新数据库的外部应用程序时，请调用内建存储过程 sp_helpconstraint，获取所有新行的列值必须遵守的所有约束的清单。然后在应用程序和存储过程中包括确保要插入的数据不违反约束的代码。

sp_helpconstraint 存储过程调用的句法如下：

```
sp_helpconstraint [@objname=]<'table name'>
[,[@nomsg=1|'nomsg']]
```

其中：

- @objname 是想让存储过程列出约束信息的表名。
- @nomsg 指定 sp_helpconstraint 是否要显示存储过程正在报告约束清单的表名。如果想压缩表名显示，可将 @NOMSG 设置为 nomsg，或者省略此参数与显示约束列表一

起显示表名。

对于每种表约束，sp_helpconstraint 返回的结果集包括：

- **CONSTRAINT_TYPE**——约束类型（CHECK、DEFAULT、PRIMARY KEY 和 FOREIGN KEY 等）以及约束应用的列。
- **CONSTRAINT_NAME**——约束的唯一用户提供的名称或者系统提供的名称。用户提供的约束名通常用来说明约束的目的。系统提供的约束名跟在约束所应用的列名之后，以随机生成的一串字母和数字结束，以保证约束名的唯一性。
- **DELETE_ACTION**——对于 FOREIGN KEY 约束来说，既可是 Cascade 也可是 No Action，而且并不适用于所有其他约束。仅当 FOREIGN KEY 定义有 ON DELETE CASCADE 规则时，DELETE_ACTION 才能是 Cascade。
- **UPDATE_ACTION**——对于 FOREIGN KEY 约束来说，既可是 Cascade 也可是 No Action，而且并不适用于所有其他约束（仅当 FOREIGN KEY 定义有 ON UPDATE CASCADE 规则时，UPDATE_ACTION 才能是 Cascade）。
- **STATUS_ENABLED**——表明不论是 FOREIGN KEY 还是 CHECK 约束都是启用的，而且不适用于所其其他约束类型。
- **STATUS_FOR_REPLICATION**——表明不论是 FOREIGN KEY 还是 CHECK 约束，在复制期间是强制实行的，而且不适用于所有其他约束类型。

除了前面的有关列约束的信息之外，sp_helpconstraint 还列出任何引用了表的 FOREIGN KEY 约束——给出每个 FOREIGN KEY 的名称以及 FOREIGN KEY 定义其上的表名。

例如，为了显示在 PUBLS 数据库中对 AUTHORS 表的约束，应执行以下语句批处理：

```
USE PUBLS
```

```
EXEC sp_helpconstraint 'authors'
```

注意：必须在想让存储过程报告约束的表的数据库内调用 sp_helpconstraint。在本例中，为了让 sp_helpconstraint 存储过程报告有关 AUTHORS 表约束，PUBLS 必须是当前数据库，如图 24.1 所示。

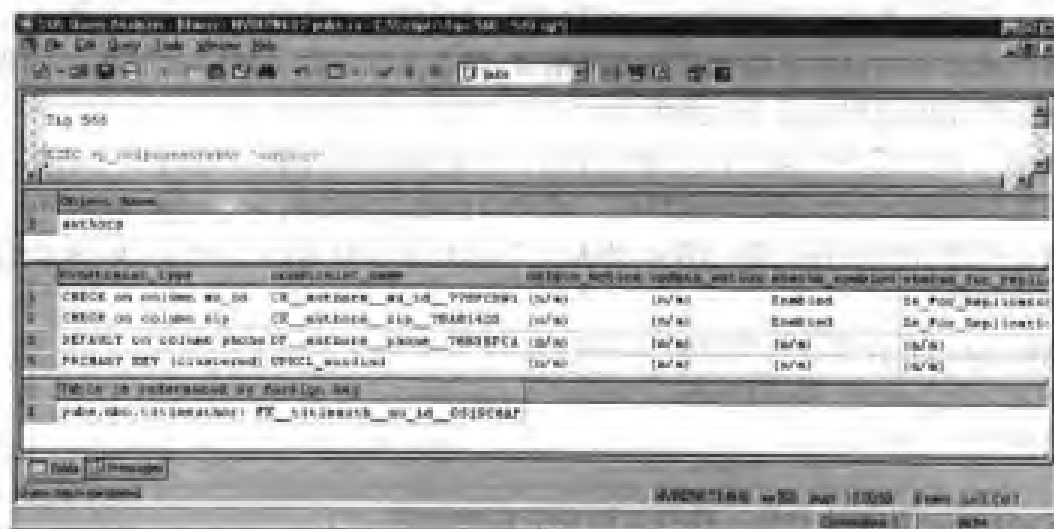


图 24.1 由存储过程 sp_helpconstraint 返回的有关 AUTHORS 表的约束信息

如要显示 CHECK 和 DEFAULT 约束的文本，可用内建的存储过程 sp_helptext（读者已在

技巧 441 “使用 `sp_helptext` 显示定义存储过程、用户定义函数、触发器、默认值、规则或者视图的文本”中学过有关知识)。在本例中，应执行以下 EXEC 命令，以显示在 AUTHORS 表内对 AU_ID 列的 CHECK 约束的文本：

```
EXEC sp_helptext 'CK__authors__au_id__77BFCB91'
```

当把系统提供的约束名传递给 `sp_helptext` 存储过程时，一定要注意约束内跟着表名 (AUTHORS) 的两处下划线，即列名 (AU_ID) 以及随机字符串 (77BFCB91)。

技巧 444 使用 `sp_pkeys` 显示表的 PRIMARY KEY 信息

通过子表的 FOREIGN KEY 引用父表内的 PRIMARY KEY，可在任意两个表之间建立父子关系。PRIMARY KEY 只不过是表内应用了 PRIMARY KEY 约束的一列或多列。类似地，FOREIGN KEY 是有 FOREIGN KEY 约束的一列或多列。PRIMARY KEY 的重要性在于，每个 PRIMARY KEY 值惟一地确定了父表内的单行。换句话说，是表的 PRIMARY KEY 列（或列集）中的每行必须是惟一的、非 NULL 值。另一方面，FOREIGN KEY 列的行内值不必是，而且对于任何子表中的行来说，事实上在理想情况下也不是惟一的。

例如，假定想要在 CUSTOMERS 表和 ORDERS 表之间建立父子关系。在这种关系中，CUSTOMERS 表中的行为“父”，而 ORDERS 表中的行为“子”。一位顾客可（而且但愿如此）不止一项定单。然而，任何特定的定单只能属于一位顾客。

为了在 CUSTOMERS 和 ORDERS 之间建立父/子关系，可用内建的存储过程 `sp_pkeys` 去确定惟一地确定 CUSTOMERS 表内的每位顾客（也就是每行）的 PRIMARY KEY 列。然后在子表 (ORDERS) 中定义引用父表 (CUSTOMERS) 内的 PRIMARY KEY 的 FOREIGN KEY 约束。

`sp_pkeys` 存储过程调用的句法如下：

```
sp_pkeys [@table_name=] '<table name>'
        [,[@table_owner=] '<table owner's username>']
        [,[@table_qualifier=] '<database name>']
```

其中：

- `@table_name` 是你想让存储过程报告有关 PRIMARY KEY 信息的表名。
- `@table_owner` 是表的所有者的用户名。如果在此过程调用中忽略了 `@table_owner` 参数，DBMS 将在所拥有的表中搜索由 `<table name>` 给出的表，然后再在由数据库所有者 (DBO) 所拥有的表中搜索。
- `@table_qualifier` 是表所在的数据库名。如果省略，那么 DBMS 假定表在当前数据库中。

这样，为了显示定义在 NORTHWIND 数据库中的 CUSTOMERS 表上的 PRIMARY KEY，可写出以下 EXEC 语句：

```
EXEC sp_pkeys @table_name='customers',
              @table_qualifier='Northwind'
```

在被调用时，`sp_pkeys` 将返回带有以下有关 PRIMARY KEY 信息的结果集：

- TABLE_QUALIFIER——表所在的数据库名。
- TABLE_OWNER——表的所有者的用户名。
- TABLE_NAME——表名。
- COLUMN_NAME——PRIMARY KEY 内的列名。

- KEY_SEQ——在 COLUMN_NAME 列内指定列的 PRIMARY KEY 内的顺序位置。
- PK_NAME——主键名。

在调用 sp_pkeys 存储过程之后，请在结果表内查看 COLUMN_NAME 列以确定 FOREIGN KEY（在本例中，在 ORDERS 表中）必须引用的父（CUSTOMERS）表内的列。然后，对子（ORDERS）表中的相应列应用 FOREIGN KEY 约束。读者在技巧 44 “使用 CREATE TABLE 语句指定外键约束”中已经学过在创建新表时如何应用 FOREIGN KEY 约束。在技巧 42 “使用 ALTER TABLE 语句改变主键和外键”中还学过如何改变现存表的结构。

如果 PRIMARY KEY 有多列，由 sp_pkeys 存储过程返回的结果集将有多行——PRIMARY KEY 内的每列对应一行。在使用多列 PRIMARY KEY 时，要特别注意 COLUMN_NAME 和 KEY_SEQ 列的值。请确保以与 PRIMARY KEY 内的列相同的顺序列出 FOREIGN KEY 内的每一列。换句话说，在 FOREIGN KEY 内的列数必须与 PRIMARY KEY 内的列数相匹配对应，而且对应列必须处在每个键字内的同样的顺序位置。KEY_SEQ 列中的数字给出了在 COLUMN_NAME 列指定的列的 PRIMARY KEY 内的顺序位置。

技巧 445 使用 sp_fkeys 显示关于引用表的 PRIMARY KEY 的外键信息

在创建关系型数据库时，一定要利用 DBMS 保持引用完整性的能力。虽然可以建立多表 SELECT 语句，联合父行和子行，而不必有 PRIMARY KEY 和 FOREIGN KEY 约束，但不要这样做。虽然为了建立 PRIMARY KEY 和 FOREIGN KEY 约束要事先花一些计划精力，但这样做可以不必关心用户是否在父表中插入重复的行或者在子表里建立了孤行。

在父表里重复的行和（或）子表里的孤行是不希望有的，因为这会导致报告的错误，还可引起实际寿命问题。例如，假定有一个 CUSTOMERS 父表和一个 ORDERS 子表。如果在 CUSTOMERS（父）表里有重复的行，那么 ORDERS 表中的子行可在 CUSTOMERS 的表里有两个（或多个）父行——这就意味着一位顾客很可能为一次交货的定单花了两次钱。相反，在子（ORDERS）表中的孤行意味着那些定单没有任何顾客信息。因此，如果 Ship To（发货到）地址储存在 ORDERS 表里，而且 Bill To（向……收款）信息被储存在 CUSTOMERS 表中，则定单的货物发出却没有顾客为此定单付款。

当建立了 PRIMARY KEY 和 FOREIGN KEY 约束之后，让 DBMS 维护相关表间的父/子表关系（也就是在数据库内保持引用完整性），人们会发现，生成引用了每个父表的 FOREIGN KEY 约束的清单是方便的。当想要删除一个父表或者改变其结构时，这个列表使用起来是便利的。此外，当执行 DELETE 语句删除父行，或者执行改变 PRIMARY KEY 列值的 UPDATE 语句时，可能还不得不删除或更新子表行中的列。当在父表里作出更改时，引用了父表的 PRIMARY KEY 的 FOREIGN KEY 列表将告诉用户需要注意哪一个子表。

内建的存储过程 sp_fkeys 可让人们得到由 MS-SQL Server 管理的数据库内的任何表上 FOREIGN_KEY 引用的列表。为了调用 sp_fkeys，可用以下句法：

```
sp_fkeys
    [@pktable_name=] '<PRIMARY KEY table name>'
    [,[@pktable_owner=] '<PRIMARY KEY table owner's name>'
    [,[@fktable_name=] '<FOREIGN KEY table name>'
    [,[@fktable_owner=] '<FOREIGN KEY table owner's name>'
    [,[@fktable_qualifier=] '<FOREIGN KEY table database>'
```

其中:

- @pktable_name 是想让存储过程列出 FOREIGN KEY 引用的父表 (带有 PRIMARY KEY) 名。
- @pktable_owner 是父表拥有者的用户名。如果在程序调用时省略了 @PKTABLE_OWNER 参数, DBMS 将在所拥有的表中搜索由 @PKTABLE_NAME 指定的表名, 然后在数据库所有者 (DBO) 拥有的表中搜索。
- @fktable_name 是子表名 (带有 FOREIGN KEY)。如果既提供了父表名 (@PKTABLE_NAME) 也提供了子表 (@FKTABLE_NAME) 名, 则存储过程将只列出一个子表 (在 @FKTABLE_NAME 参数中指定的) 内对父表 (在 @PKTABLE_NAME 参数中指定的) 的 FOREIGN KEY 引用。相反, 如果只提供了子表 (在 @FKTABLE_NAME 参数中) 名而忽略了父表名, 存储过程将列出子表内由外键所引用的所有父表。
- @fktable_owner 是子表所有者的用户名。如果省略了 @FKTABLE_OWNER 参数, DBMS 先在所拥有的表中搜索由 @FKTABLE_NAME 指定的子表, 然后再在数据库所有者 (DBO) 拥有的表中搜索子表。
- @fktable_qualifier 是子表 (带有 FOREIGN KEY 约束) 所在的数据库名。如果省略, DBMS 假定子 (FOREIGN KEY) 表存在于当前的数据库。

这样一来, 如果想要列出在引用了 NORTHWIND 数据库里的 CUSTOMERS 表内的 PRIMARY KEY 所有 FOREIGN KEY 约束的列表, 可执行以下语句批处理:

```
USE Northwind
EXEC sp_fkeys @pktable_name='Customers'
```

接着, 存储过程 sp_fkeys 将返回每个 FOREIGN KEY 引用至少一行的结果集, 而且每行中有以下各列:

- PKTABLE_QUALIFIER——带有 PRIMARY KEY 约束的 (父) 表所在的数据库名。
- PKTABLE_OWNER——父表拥有者的用户名。
- PKTABLE_NAME——带有 PRIMARY KEY 约束的 (父) 表名。
- PKCOLUMN_NAME——PRIMARY KEY 中的列名。如果 PRIMARY KEY 有多列, sp_fkeys 将返回多行——PRIMARY KEY 约束内的每列都有一行。
- FKTABLE_QUALIFIER——带有 FOREIGN KEY 约束的子表所在的数据库表名。
- FKTABLE_OWNER——子表所有者的用户名。
- FKTABLE_NAME——带有 FOREIGN KEY 约束的 (子) 表名。
- FKCOLUMN_NAME——与 PKCOLUMN_name (在结果集的当前行) 指定的 PRIMARY KEY 列对应的 FOREIGN KEY 内的列名。
- KEY_SEQ——结果表内由当前行所描述的 PRIMARY KEY 和 FOREIGN KEY 列内的顺序位置。
- UPDATE_RULE——当 PRIMARY KEY 内相应列值被更新时, 指定 DBMS 对 FOREIGN KEY 列内的值所采取的行动。其值为 0、1、2。0 = Cascade、1 = No Action、2 = Set Null。
- DELETE_RULE——当父表内子表的相应行中的值被删除时, 指明 DBMS 对

FOREIGN KEY 列内的值所采取的行动。其值可为 0、1 或 2。0 = Cascade; 1 = No Action 和 2 = Set Null。

- FK_NAME——用户或系统提供的 FOREIGN KEY 约束的名称。
- PK_NAME——PRIMARY KEY 约束的用户或系统提供的名。

为了显示由子表内的 FOREIGN KEY 引用的所有父表的以上列出的信息，可调用 sp_fkey，并只向其提供有关子表（即有 FOREIGN KEY 约束的）的信息。例如，为了列出 NORTHWIND 数据库中的 ORDERS 表内的 FOREIGN KEY 约束所引用的所有表的清单，可执行以下语句批处理：

```
USE Northwind
EXEC sp_fkeys @fktable_name='Orders'
```

技巧 446 使用 sp_procoption 控制 MS-SQL Server 启动时运行的存储过程

在技巧 411~技巧 413 中，读者已经学习了存储过程能做什么，如何使用 CREATE PROCEDURE 语句建立存储过程以及如何使用 Transact-SQL EXEC 命令调用存储过程。简而言之，存储过程就是一组 SQL 和（或）Transact-SQL 语句，无论何时，只要用户通过名称调用存储过程，DBMS 就要执行这些语句。存储过程是强大的功能，因为它们允许编写 SQL 程序。也就是说，存储过程允许人们把 SQL 数据管理语句与过程化的 Transact-SQL 编程语句结合起来，从而可创建执行复杂查询、数据库更新的语句批处理，或者执行想要执行的任何 SQL（和 Transact-SQL 语句）序列。存储过程使得数据库用户有可能通过提交单条调用存储过程的 EXEC 语句，执行复杂的查询或者执行大量的数据库管理语句批处理，然后执行所要求的工作。

除了让用户和外部应用程序执行存储过程之外，可在每次启动 DBMS 时，让 MS-SQL Server 自动地执行特定的存储过程。例如，如果有一组维护任务，如创建或清除全局临时表或游标，创建行政总结表，或者某些其他的清理/预备性的任务，每当启动 MS-SQL Server 时，都想让 DBMS 来执行，可将执行工作的语句放入一个或几个存储过程。然后，用内建的存储过程 sp_procoption 来标记启动时就要调用的存储过程。

为了创建 DBMS 在启动将执行的存储过程，必须以数据库所有者（DBO）身份登录 MASTER 数据库。作为 DBO，并在 MASTER 数据库内，可使用 CREATE PROCEDURE 语句创建想让 DBMS 在启动时执行的存储过程。接着，用以下句法调用 sp_procoption 存储过程以标记所创建并在启动时执行的存储过程：

```
sp_procoption [@procname=] '<stored procedure name>'
               ,[@optionname=] 'startup'
               ,[@optionvalue=] '{true|false}'
```

其中：

- @procname 是想让 MS-SQL 在启动时就运行的存储过程名。或者，如果把 @OPTIONVALUE 设为 FALSE，@PROCNAME 可以是不再想让 MS-SQL Server 在启动时就运行的存储过程名。
- @optionname 总是 startup（启动）。
- @optionvalue 既可为 TRUE 也可为 FALSE。如果想让 DBMS 在启动时执行在 @PROCNAME 中指定的过程，应把 @OPTIONVALUE 设置为 TRUE。如果不再想让 DBMS 在启动时就执行程序，就把 @OPTIONVALUE 设置为 FALSE。

例如，假定以 DBO 身份创建了 MASTER 数据库内的存储过程 su_sp_CreateExecSummary。

为了每当启动 MS-SQL Server 时, 让 DBMS 调用 `su_sp_CreateExecSummary`, 可执行以下语句批处理:

```
USE master
EXEC sp_procoption @procname = 'su_sp_CreateExecSummary',
                   @optionname = 'startup',
                   @optionvalue = 'true'
```

注意: 由于 MS-SQL Server 不允许在一个存储过程中执行 USE 语句, 必须在让 DBMS 执行启动时执行的存储过程中使用形式为 `<database name>.<owner ID>.<table name>` 的完全合格的名称引用数据库对象。

为了使跟踪在启动时执行的存储过程变得容易一些, 可给存储过程起一个以 `su_sp_` 开头的描述性名称, 这个开头代表 startup stored procedure (启动时的存储过程)。

当不再想让 MS-SQL Server 在启动时执行存储过程时, 像以前一样调用 `sp_procoption` 存储过程, 只是这一次要把 `@OPTIONVALUE` 设置为 FALSE, 如下所示:

```
USE master
EXEC sp_procoption @procname = 'su_sp_CreateExecSummary',
                   @optionname = 'startup',
                   @optionvalue = 'false'
```

在本例中, DBMS 将对存储过程 `su_sp_CreateExecSummary` 禁用启动时的执行选项。

技巧 447 使用 `sp_spaceused` 显示分配给数据库或单独的数据库对象的已用与未用空间量

在技巧 421 “理解 DBCC 的维护语句”中, 读者已经学习了如何使用 DBCC 的 `SHRINKDATABASE` 和 `SHRINKFILE` 来减少数据库数据 (.mdf) 以及事务处理日志 (.ldf) 文件的容量。这两个 DBCC 选项涉及从数据库数据和事务处理日志中删除未使用的空间, 然后将未使用的空间返回给操作系统控制。

数据库数据 (.mdf) 文件有时有太多的空白/未使用的空间, 因为 DBMS 随着向数据库内的表中添加数据会扩展其文件容量。除非指定自动收缩选项, 否则 DBMS 不减少其文件在磁盘上的容量, 即使从数据库表中删除了大量数据或较大的索引也是如此。DBMS 不是将以前由数据所占空间释放给操作系统, 而是在其数据文件中保留着未用空间, 只是将该处空间标示为可用的, 以后用户可向其中添加数据。

用户可为数据库设置自动收缩选项, 但是, 并不提倡这样做, 因为这或许会增加不必要的开销。例如, 当数据库文件内未用的磁盘空间由于某种目的不再需要时, DBMS 有可能强制收缩然后重新增大数据库文件。

内建的存储过程 `sp_spaceused` 将告诉用户在数据库文件中有多少已用和未用空间。事实上, 可以用以下句法调用该存储过程, 以便获得整个数据库或是数据库内单独的表的空间使用率信息:

```
sp_spaceused [[@objname='<table name>']
             [,[@updateusage='updateusage']]
```

其中:

- `@objname` 是想要让例程显示其空间使用率数据的表的名称。如果忽略了

@OBJNAME 参数，该存储过程就报告当前数据库的空间使用率信息（而不是对数据库内单独的表）。

- @updateusage 指定 DBMS 是否必须扫描磁盘上的数据和索引页，以便获得实际的磁盘使用率数据并改正 SYSINDEXES 表中的使用率数字。通常，如果最近从数据库中删除过较大的索引，则人们只想指定@UPDATEUSAGE 选项，因为 SYSINDEXES 表可能还没有最新的空间使用率信息结果。

例如，为了确定 SQLTips 数据库数据文件 (.mdf) 以及留在文件内的未分配空间的总容量（以 MB 计），可向 DBMS 提交以下 EXEC 语句：

```
EXEC sp_spaceused
```

接着，MS-SQL Server 将显示有下面各列的结果集：

- DATABASE_NAME——当前数据库的名称。
- DATABASE_SIZE——当前数据库数据文件的总容量。
- UNALLOCATED_SPACE——还没有分配给数据库对象的空间量（以 MB 计）。
- RESERVED——分配给指定给数据库内还未填充数据的表和索引的 8KB 页面的空间。
- DATA——用于保存所有表内数据的空间量。
- INDEX_SIZE——用于保存数据库内所有索引数据的空间量。
- UNUSED——数据库文件内还未指定给任何数据库对象的空间量。

如果如下所示调用 sp_spaceused 存储过程而且提供了当前数据库内表的名称，sp_spaceused 只报告该表的使用率统计数字：

```
USE pubs
```

```
EXEC sp_spaceused @objname='authors'
```

在本例中，DBMS 将报告 PUBLS 数据库内 AUTHORS 表的使用率信息：

- NAME——所请求报告空间使用率的表名。
- ROWS——表内数据的行数。
- RESERVED——分配给指定给数据库内还未填充数据的表和索引的 8Kb 页面的空间。
- DATA——用于保存所有表内数据的空间量。
- INDEX_SIZE——用于保存数据库内所有索引数据的空间量。
- UNUSED——已分配给表但还未作为 8Kb 数据页的一部分而使用的空间。

技巧 448 使用 sp_helptrigger 显示有关表上的触发器信息

正如读者在技巧 348~技巧 351 中所学习的，触发器是一种 DBMS 在对特定表响应 INSERT、UPDATE 或 DELETE 操作时所执行的特殊类型的存储过程。例如，在执行了 CREATE TRIGGER 语句之后，在某个表上创建了一个 INSERT 触发器（TRIGGER），DBMS 将在对象为该表的所有 INSERT 语句之前、之后或对象为该表的非 INSERT 语句从而“触发”（也就是执行）触发器内的语句。在创建触发器时，作为选项之一可指定 DBMS 是在 INSERT、UPDATE 或 DELETE 语句之前或之后，还是不是这些语句时“触发”触发器。

除了 DBMS 自动执行触发器（以响应 INSERT、UPDATE 或 DELETE 语句，根据触发器的类型不同而响应不同的语句），触发器看起来与存储过程完全相同。不幸的是，人们在列出表上的普通存储过程（非触发器）时看不到触发器。在 MS-SQL Server Enterprise Manager 内，

通过单击数据库的 Stored Procedures（存储过程）图标可获得在数据库内定义的非触发器存储过程的清单。

另外，虽然内建的存储过程 `sp_help` 显示有关 FOREIGN KEY 约束（其中可能有 ON DELETE 或 ON UPDATE 规则），但 `sp_help` 不会告诉人们任何有关在表上定义的触发器的内容。结果，在 MS-SQL Server 上，对在每个数据库内的表上定义的所有触发器都必须保持最新最全的文档。如果不如此，服务器看起来好像在响应某些（但不是全部）INSERT、UPDATE 和 DELETE 语句时会采取随机的行动。

可按以下句法调用内建存储过程 `sp_helptrigger`，以便列出在数据库表上定义的触发器：

```
sp_helptrigger [@tabname=] '<table name>'
               [,[@triggertype=] '{DELETE|INSERT|UPDATE}']
```

其中：

- `@tabname` 当前数据库内想要该存储过程提供所定义的触发器的表名。
- `@triggertype` 指定存储过程要提供信息的触发器的类型。如果忽略了 `@TRIGGER_TYPE`，则 `sp_helptrigger` 将返回有关在该表上定义的所有触发器的信息。

对于每个触发器（或对于由 `@TRIGGER_TYPE` 参数所指定的类型的触发器），`sp_helptrigger` 返回包括以下各列的结果集：

- TRIGGER_NAME——触发器名称。
- TRIGGER_OWNER——触发器拥有者的用户名。
- ISUPDATE——如果是 UPDATE 触发器则为 1，否则为 0。
- ISDELETE——DELETE 触发器则为 1，否则为 0。
- ISINSERT——如果是 INSERT 触发器则为 1，否则为 0。
- ISAFTER——如果 DBMS 在执行触发语句 UPDATE、DELETE 或 INSERT 之后（而不是之前或不是这些语句）执行触发器，其值为 1，否则为 0。
- ISINSTEADOF——如果 DBMS 在执行的不是触发语句 UPDATE、DELETE 或 INSERT 时（而不是之前或之后）执行触发器，其值为 1，否则为 0。

这样一来，为了显示有关在 SQLTips 数据库内的 EMPLOYEES 表上定义的所有触发器的信息，可执行以下语句批处理：

```
USE SQLTips
EXEC sp_helptrigger 'employees'
```

或者，如果 SQLTips 已经是当前数据库，只要调用 `sp_helptrigger` 存储过程而不必执行 USE 语句。例如，为了只显示 EMPLOYEES 表上的 UPDATE 触发器，可使用以下 EXEC 语句：

```
EXEC sp_helptrigger 'employees', 'update'
```

注意：为了省心，请保持列出并描述由 MS-SQL Server 管理的数据库内每个表上定义的触发器最新文档。如果不这样做，DBMS 看起来好像在做奇怪的事情。请记住，每个数据库可能会有上百个表，而且同时会有几十个人在提交 INSERT、UPDATE 和 DELETE 语句。因而，试图查明引起似乎不确定的 DBMS 行为的特定 INSERT、UPDATE 或 DELETE 语句并不是一件小事。正如在本技巧中所学习的，`p_helptrigger` 存储过程要求至少知道触发器所定义其上的表名。

如果丢失了文档，可执行对 SYSOBJECTS 表的以下查询，从而获得当前数据库内的表上定义的触发器的清单：

```
SELECT * FROM sysobjects WHERE type = 'TR'
```

请查看一下对应定义于数据库内表上的触发器名的结果集的行内的 NAME 列。接着执行下面的语句（用从结果集中得来的触发器名代替以下语句中的 <name of trigger> 部分），查看触发器定义，其中包括该触发器定义其上的表名：

```
EXEC sp_helptext <name of trigger>
```

技巧 449 使用 sp_who 和 KILL 命令控制运行在 MS-SQL Server 上的进程

有时要求所有用户都从数据库上注销，才能执行定时维护任务或非定时的修理工作。例如，为了运行在技巧 424 “理解 DBCC 的确认语句”中所学习的 DBCC 语句，如果想要运行带修理选项的 DBCC 确认任务，必须首先将数据库改变为单用户模式。为了从多用户模式改变为单用户模式，要求所有人都从数据库上注销。类似地，如果想要将数据库文件从一个驱动器转移到另一驱动器上，这也要求所有用户都必须从数据库上注销。接着，就可断开、移动到新磁盘上，然后重新连接数据库文件（读者在技巧 438 “使用 sp_detach_db 和 sp_attach_db 在 MS-SQL Server 上删除和添加数据库”中学习了如何断开并重新连接数据库文件）。

在向所有工作人员宣布请其从系统上注销之后，必须检查运行在 MS-SQL Server 上的进程的清单，从而找出每个人何时确实已经注销了。为了获得登录到 MS-SQL Server 上的用户和应用程序的清单，可调用内建的存储过程 sp_who，请使用如下句法：

```
EXEC sp_who [[@loginame=] '<login name>']
```

其中：

- @loginame 是想让该存储过程提供 SPID 和其他进程状态信息的与进程或登录相关人员的用户名。

这样一来，为了显示正在运行的当前进程和登录到 MS-SQL Server 上的用户的清单，可提交如下 EXEC 语句显示如图 24.2 所示的进程清单。

spid	status	session_id	loginame	hostname	dbid	dbname	text
1	0	background	sa		0	NULL	LAZY WRITER
2	0	background	sa		0	NULL	LOG WRITER
3	0	background	sa		0	master	SIGNAL HANDLER
4	0	background	sa		0	NULL	LOCK MONITOR
5	0	background	sa		0	tempdb	TASK MANAGER
6	0	background	sa		0	master	TASK MANAGER
7	0	background	sa		0	NULL	CHECKPOINT SLEEP
8	0	background	sa		0	master	TASK MANAGER
9	0	background	sa		0	master	TASK MANAGER
10	0	background	sa		0	master	TASK MANAGER
11	0	background	sa		0	master	TASK MANAGER
12	0	background	sa		0	master	TASK MANAGER
13	0	background	sa		0	master	TASK MANAGER
14	0	background	sa		0	master	TASK MANAGER
15	0	background	sa		0	master	TASK MANAGER
16	0	background	sa		0	master	TASK MANAGER
17	0	background	sa		0	master	TASK MANAGER
18	0	background	sa		0	master	TASK MANAGER
19	0	background	sa		0	master	TASK MANAGER
20	0	background	sa		0	master	TASK MANAGER
21	0	background	sa		0	master	TASK MANAGER
22	0	background	sa		0	master	TASK MANAGER
23	0	background	sa		0	master	TASK MANAGER
24	0	background	sa		0	master	TASK MANAGER
25	0	background	sa		0	master	TASK MANAGER
26	0	background	sa		0	master	TASK MANAGER
27	0	background	sa		0	master	TASK MANAGER
28	0	background	sa		0	master	TASK MANAGER
29	0	background	sa		0	master	TASK MANAGER
30	0	background	sa		0	master	TASK MANAGER
31	0	background	sa		0	master	TASK MANAGER
32	0	background	sa		0	master	TASK MANAGER
33	0	background	sa		0	master	TASK MANAGER
34	0	background	sa		0	master	TASK MANAGER
35	0	background	sa		0	master	TASK MANAGER
36	0	background	sa		0	master	TASK MANAGER
37	0	background	sa		0	master	TASK MANAGER
38	0	background	sa		0	master	TASK MANAGER
39	0	background	sa		0	master	TASK MANAGER
40	0	background	sa		0	master	TASK MANAGER
41	0	background	sa		0	master	TASK MANAGER
42	0	background	sa		0	master	TASK MANAGER
43	0	background	sa		0	master	TASK MANAGER
44	0	background	sa		0	master	TASK MANAGER
45	0	background	sa		0	master	TASK MANAGER
46	0	background	sa		0	master	TASK MANAGER
47	0	background	sa		0	master	TASK MANAGER
48	0	background	sa		0	master	TASK MANAGER
49	0	background	sa		0	master	TASK MANAGER
50	0	background	sa		0	master	TASK MANAGER
51	0	background	sa		0	master	TASK MANAGER
52	0	background	sa		0	master	TASK MANAGER
53	0	background	sa		0	master	TASK MANAGER
54	0	background	sa		0	master	TASK MANAGER
55	0	background	sa		0	master	TASK MANAGER
56	0	background	sa		0	master	TASK MANAGER
57	0	background	sa		0	master	TASK MANAGER
58	0	background	sa		0	master	TASK MANAGER
59	0	background	sa		0	master	TASK MANAGER
60	0	background	sa		0	master	TASK MANAGER
61	0	background	sa		0	master	TASK MANAGER
62	0	background	sa		0	master	TASK MANAGER
63	0	background	sa		0	master	TASK MANAGER
64	0	background	sa		0	master	TASK MANAGER
65	0	background	sa		0	master	TASK MANAGER
66	0	background	sa		0	master	TASK MANAGER
67	0	background	sa		0	master	TASK MANAGER
68	0	background	sa		0	master	TASK MANAGER
69	0	background	sa		0	master	TASK MANAGER
70	0	background	sa		0	master	TASK MANAGER
71	0	background	sa		0	master	TASK MANAGER
72	0	background	sa		0	master	TASK MANAGER
73	0	background	sa		0	master	TASK MANAGER
74	0	background	sa		0	master	TASK MANAGER
75	0	background	sa		0	master	TASK MANAGER
76	0	background	sa		0	master	TASK MANAGER
77	0	background	sa		0	master	TASK MANAGER
78	0	background	sa		0	master	TASK MANAGER
79	0	background	sa		0	master	TASK MANAGER
80	0	background	sa		0	master	TASK MANAGER
81	0	background	sa		0	master	TASK MANAGER
82	0	background	sa		0	master	TASK MANAGER
83	0	background	sa		0	master	TASK MANAGER
84	0	background	sa		0	master	TASK MANAGER
85	0	background	sa		0	master	TASK MANAGER
86	0	background	sa		0	master	TASK MANAGER
87	0	background	sa		0	master	TASK MANAGER
88	0	background	sa		0	master	TASK MANAGER
89	0	background	sa		0	master	TASK MANAGER
90	0	background	sa		0	master	TASK MANAGER
91	0	background	sa		0	master	TASK MANAGER
92	0	background	sa		0	master	TASK MANAGER
93	0	background	sa		0	master	TASK MANAGER
94	0	background	sa		0	master	TASK MANAGER
95	0	background	sa		0	master	TASK MANAGER
96	0	background	sa		0	master	TASK MANAGER
97	0	background	sa		0	master	TASK MANAGER
98	0	background	sa		0	master	TASK MANAGER
99	0	background	sa		0	master	TASK MANAGER
100	0	background	sa		0	master	TASK MANAGER
101	0	background	sa		0	master	TASK MANAGER
102	0	background	sa		0	master	TASK MANAGER
103	0	background	sa		0	master	TASK MANAGER
104	0	background	sa		0	master	TASK MANAGER
105	0	background	sa		0	master	TASK MANAGER
106	0	background	sa		0	master	TASK MANAGER
107	0	background	sa		0	master	TASK MANAGER
108	0	background	sa		0	master	TASK MANAGER
109	0	background	sa		0	master	TASK MANAGER
110	0	background	sa		0	master	TASK MANAGER
111	0	background	sa		0	master	TASK MANAGER
112	0	background	sa		0	master	TASK MANAGER
113	0	background	sa		0	master	TASK MANAGER
114	0	background	sa		0	master	TASK MANAGER
115	0	background	sa		0	master	TASK MANAGER
116	0	background	sa		0	master	TASK MANAGER
117	0	background	sa		0	master	TASK MANAGER
118	0	background	sa		0	master	TASK MANAGER
119	0	background	sa		0	master	TASK MANAGER
120	0	background	sa		0	master	TASK MANAGER
121	0	background	sa		0	master	TASK MANAGER
122	0	background	sa		0	master	TASK MANAGER
123	0	background	sa		0	master	TASK MANAGER
124	0	background	sa		0	master	TASK MANAGER
125	0	background	sa		0	master	TASK MANAGER
126	0	background	sa		0	master	TASK MANAGER
127	0	background	sa		0	master	TASK MANAGER
128	0	background	sa		0	master	TASK MANAGER
129	0	background	sa		0	master	TASK MANAGER
130	0	background	sa		0	master	TASK MANAGER
131	0	background	sa		0	master	TASK MANAGER
132	0	background	sa		0	master	TASK MANAGER
133	0	background	sa		0	master	TASK MANAGER
134	0	background	sa		0	master	TASK MANAGER
135	0	background	sa		0	master	TASK MANAGER
136	0	background	sa		0	master	TASK MANAGER
137	0	background	sa		0	master	TASK MANAGER
138	0	background	sa		0	master	TASK MANAGER
139	0	background	sa		0	master	TASK MANAGER
140	0	background	sa		0	master	TASK MANAGER
141	0	background	sa		0	master	TASK MANAGER
142	0	background	sa		0	master	TASK MANAGER
143	0	background	sa		0	master	TASK MANAGER
144	0	background	sa		0	master	TASK MANAGER
145	0	background	sa		0	master	TASK MANAGER
146	0	background	sa		0	master	TASK MANAGER
147	0	background	sa		0	master	TASK MANAGER
148	0	background	sa		0	master	TASK MANAGER
149	0	background	sa		0	master	TASK MANAGER
150	0	background	sa		0	master	TASK MANAGER
151	0	background	sa		0	master	TASK MANAGER
152	0	background	sa		0	master	TASK MANAGER
153	0	background	sa		0	master	TASK MANAGER
154	0	background	sa		0	master	TASK MANAGER
155	0	background	sa		0	master	TASK MANAGER
156	0	background	sa		0	master	TASK MANAGER
157	0	background	sa		0	master	TASK MANAGER
158	0	background	sa		0	master	TASK MANAGER
159	0	background	sa		0	master	TASK MANAGER
160	0	background	sa		0	master	TASK MANAGER
161	0	background	sa		0	master	TASK MANAGER
162	0	background	sa		0	master	TASK MANAGER
163	0	background	sa		0	master	TASK MANAGER
164	0	background	sa		0	master	TASK MANAGER
165	0	background	sa		0	master	TASK MANAGER
166	0	background	sa		0	master	TASK MANAGER
167	0	background	sa		0	master	TASK MANAGER
168	0	background	sa		0	master	TASK MANAGER
169	0	background	sa		0	master	TASK MANAGER
170	0	background	sa		0	master	TASK MANAGER
171	0	background	sa		0	master	TASK MANAGER
172	0	background	sa		0	master	TASK MANAGER
173	0	background	sa		0	master	TASK MANAGER
174	0	background	sa		0	master	TASK MANAGER
175	0	background	sa		0	master	TASK MANAGER
176	0	background	sa		0	master	TASK MANAGER
177	0	background	sa		0	master	TASK MANAGER
178	0	background	sa		0	master	TASK MANAGER
179	0	background	sa		0	master	TASK MANAGER
180	0	background	sa		0	master	TASK MANAGER
181	0	background	sa		0	master	TASK MANAGER
182	0	background	sa		0	master	TASK MANAGER
183	0	background	sa		0	master	TASK MANAGER
184	0	background	sa		0	master	TASK MANAGER
185	0	background	sa		0	master	TASK MANAGER
186	0	background	sa		0	master	TASK MANAGER
187	0	background	sa		0	master	TASK MANAGER
188	0	background	sa		0	master	TASK MANAGER
189	0	background	sa		0	master	TASK MANAGER
190	0	background	sa		0	master	TASK MANAGER
191	0	background	sa		0	master	TASK MANAGER
192	0	background	sa		0	master	TASK MANAGER
193	0	background	sa		0	master	TASK MANAGER
194	0	background	sa		0	master	TASK MANAGER
195	0	background	sa		0	master	TASK MANAGER
196	0	background	sa		0	master	TASK MANAGER
197	0	background	sa		0	master	TASK MANAGER
198	0	background	sa		0	master	TASK MANAGER
199	0	background	sa		0	master	TASK MANAGER
200	0	background	sa		0	master	TASK MANAGER
201	0	background	sa		0	master	TASK MANAGER
202	0	background	sa		0	master	TASK MANAGER
203	0	background	sa		0	master	TASK MANAGER
204	0	background	sa		0	master	TASK MANAGER
205	0	background	sa		0	master	TASK MANAGER
206	0	background	sa		0	master	TASK MANAGER
207	0	background	sa		0	master	TASK MAN


```
EXEC sp_who
```

为了确定要结束哪个进程，可检查要让所有用户都注销的数据库名称所对应的 DBNAME 列。记下在 SPID 列内显示的每个用户的系统进程 ID。如果用户没有在预先定好的时间内注销，可向 Transact-SQL 的 KILL 命令提供该 SPID，以便将该用户的会话结束。

为了结束一个用户的连接，可使用如下句法执行 Transact-SQL 的 KILL 命令：

```
EXEC KILL {spid}[WITH STATUSONLY]
```

其中：

- @spid 是想要结束的会话的系统进程 ID。
- WITH STATUSONLY 指定 DBMS 提供状态报告只显示撤消在未提交的事务处理中已经完成的工作将要花费的时间。

例如，为了结束用户 FRANK 在 MS-SQL Server 上的连接（在前面的图 24.2 中显示 SPID 为 53），应执行以下 KILL 命令：

```
KILL 53
```

在使用 KILL 结束个连接时，DBMS 必须取消任何未提交的由连接所完成的工作。虽然 DBMS 处于撤消所完成的未提交工作的过程中，此进程仍然处于由 sp_who 显示的进程清单中，但是，连接状态将显示为 KILLED/ROLLBACK。

为了获得撤消过程所花时间有多长的概念，可执行带有 WITH STATUSONLY 选项的 KILL 命令，如下所示：

```
KILL 53 WITH STATUSONLY
```

MS-SQL Server 接着将以如下消息加以响应：

```
Spid 53: Transaction rollback in progress. Estimated  
rollback completion: 75% Estimated time left: 30 seconds.
```

除了结束进程，使得每个用户都从数据库中退出，可能还要使用 KILL 来结束执行不确定查询或其锁定正在阻塞其他进程的进程，此内容将在下一技巧中加以讨论。

技巧 450 使用 sp_lock 显示数据库所掌握的锁定信息

在技巧 266~技巧 269 中，读者学习了有关事务处理隔离级别方面的内容。总结起来说，3 种隔离级别（SERIALIZABLE、REPEATABLE READ 和 READ COMMITTED）中的每一种都意味着防止某种更新方面的错误并当两个或多个用户并发地更新并查询同一表时报告出现的错误。最高级别的 SERIALIZABLE，可防止所有的更新并通过给予一位用户以对所读取的、更新的或是插入的数据的专门锁定而报告错误。在用户关闭 SERIALZABLE 事务处理之前，用户保持着对一个或多个表内所读取或改变的数据的专门锁定，其他想要对锁定的页面内的数据执行 INSERT 或 UPDATE 语句的用户必须等待，直到有专门锁定的用户结束了事务处理（使用 COMMIT 或 ROLLBACK 语句）或者降低隔离级别，以便允许共享访问。

如果一个用户已经等待完成 INSERT 或 UPDATE 语句很长时间了，可使用内建的存储过程 sp_lock 来确定哪一位其他用户（或进程）正在掌握着对表（或是表内的页面）的专门锁定。在使用 sp_lock 确定了锁定的会话的系统进程 ID（SPID）之后，如果必要可使用 KILL 命令结束锁定数据的会话。

调用 sp_lock 存储过程的句法如下：

```
sp_lock [(@spid1=] '<a SPID>',[, [(@spid2=] '<a second SPID>']
```

其中：

- @spid1 是想要显示锁定信息的进程的 SPID。
- @spid2 是想要显示锁定信息的第二进程的 SPID。

这样一来，可指定 0 个、一个或两个参数来调用 sp_lock。如果提供 0 个参数（也就是调用时没有提供什么参数），该存储过程将显示当前数据库内所有会话的锁定信息。

另外，如果提供了 @SPID1 参数，sp_lock 将报告一个会话的锁定信息，而如果提供了 @SPID1 和 @SPID2 两者，则存储过程将报告两个会话的位置。

为了解决锁定问题，首先可使用内建的存储过程 sp_who 来确定被锁定会话的 SPID。例如，假定 Frank 报告，他已经等待了几个小时（至少对于 Frank 的感觉来说是那样），INSERT 语句还没有完成。为了确定 Frank 的 SPID，可执行下面的语句：

```
EXEC sp_who 'frank'
```

当 sp_who 返回其结果集时，可查看一下 SPID 列并记下 Frank 的 SPID，例如 54。

接着执行下面的对 sp_lock 存储过程的调用，来显示所有具有由 DBMS 授予的打开锁定的 SPID 和锁定状态：

```
EXEC sp_lock
```

当存储过程 sp_lock 返回类似于图 24.3 所示的结果集时，在结果集的 SPID 列中找出 Frank 的 SPID（在本例中是 54）。接着横着查看所有带 Frank 的 SPID 的行的右边，在 STATUS 列中找出带有 WAIT 字样的一行。

	spid	dbid	Objid	Indid	Type	Resource	Mode	Status
24	51	1	85575343	2	KEY	(0701d0ff1b4e)	RangeS-S	GRANT
25	51	1	85575343	2	KEY	(ce00dd36d9c0)	RangeS-S	GRANT
26	51	1	85575343	2	KEY	(bc00d6b55f46)	RangeS-S	GRANT
27	51	7	2101582525	0	TAB		IX	GRANT
28	51	1	85575343	0	TAB		IS	GRANT
29	51	1	85575343	2	KEY	(1101ed75e828)	RangeS-S	GRANT
30	51	1	85575343	3	KEY	(7f00d0c0506b)	RangeS-S	GRANT
31	52	7	0	0	DB		S	GRANT
32	53	7	0	0	DB		S	GRANT
33	53	1	85575343	0	TAB		IS	GRANT
34	54	7	2101582525	0	TAB		IX	WAIT

图 24.3 sp_lock 内建存储过程调用的输出

在 STATUS 列中找出带有 WAIT 字样的一行之后，返回去查看该行的左边并记下 DBID 列中的对象 ID。在 DBID 列中向上或向下移动，直到找到另一进程在该列使用同一 ID。

在这个示例中，SPID 51 正在使用 DBID 7，这是与 Frank 正在“等待”的连接一样的 DBID（正如图 24.3 所示）。既然有了两个明显冲突的 SPID，可再次调用 sp_lock，提供两个 SPID，以便确认一个确实锁定了另一个想要获得的资源：

```
EXEC sp_lock 51, 53
```

在看到一用户（在本例中是 51）确实给予对表（或其他资源）的锁定（由行的 STATUS 列中的 GRANT 字样而指明的）而另一正在等待时，既可与保持锁定的用户联系，也可简单地结束该 SPID。为了确定保持锁定的用户名，可调用 sp_who 存储过程并向其传递想获得用户名的 SPID，如下所示：

```
EXEC sp_who 51
```

当存储过程返回结果集时，请查看想要结束会话的 LOGINAME 列以便找出其用户名。接着，体面地给该用户打个电话。有可能保持对资源锁定的用户正处于长时间的更新或查询过程中，或者他（或她）有需要对表的专有访问的合法原因。

如果必要，可执行如下的 KILL 命令，结束锁定进程（让 Frank 的会话争取到对表的锁定并执行其 INSERT 语句）：

```
KILL 51
```

在本例中，锁定会话的 SPID 是 51，KILL 命令或许有完全不同的 SPID。

技巧 451 使用 sp_password 改变账户密码

在技巧 99 “使用 MS-SQL Server Enterprise Manager 添加登录和用户”中，读者学习了如何给予用户以对 MS-SQL Server 的访问权。当授予一个用户名以数据库访问权时，必须选择 Windows 认证或 SQL Server 认证两者之一。

如果选择了 Windows 认证，MS-SQL Server 将不再询问指定新的用户名和密码。DBMS 依靠 WindowsNT/2000/XP 操作系统来对用户的登录认证。登录 Windows 网络之后，该用户就可登录到 MS-SQL Server 上，只要告诉 DBMS 使用 Windows 认证的用户名即可。MS-SQL Server 提取用户登录 Windows 网络的用户名并对照合法用户名加以检查。如果在 Windows 认证允许登录的用户名清单中找到了这一用户名，MS-SQL Server 就允许其登录到 DBMS。

如果选择了 SQL 服务器认证，MS-SQL Server 提示输入新创建的用户名的密码。当选择 SQL 服务器认证时，DBMS 并不依靠 WindowsNT/2000/XP 操作系统来对用户的登录认证。而是，只要用户试图登录，DBMS 就要求用户输入与登录用户名匹配的密码。在用户输入用户名和密码之后，DBMS 对照其访问列表检查用户名/密码对。如果在访问列表中找到了用户名/密码对，则 MS-SQL Server 让该用户登录到 DBMS。

当用户使用 MS-SQL Server Enterprise Manager 或是内建的存储过程来创建使用 SQL 服务器认证的账户时，要改变账户密码的惟一方法是，调用内建的存储过程 sp_password。请注意，在创建 Windows 认证的账户时，WindowsNT/2000/XP 操作系统（不是 MS-SQL Server）保存着每个账户的密码。因而，为了改变 Windows 网络账户的密码，用户必须使用 Windows 网络工具或是与网络系统管理员联系。简短地说，MS-SQL Server 只能改变 MS-SQL Server 认证的密码，而这样做也只能通过存储过程 sp_password。

为了改变 MS-SQL Server 认证账户的密码，可使用如下句法调用 sp_password 存储过程：

```
sp_password [[@old=] '<the current password>']  
            ([@new] '<the new password>')  
            [,[@loginame=] '<username>']
```

其中：

- @old 是该账户的当前密码。任何用户都可改变其密码，但必须提供该账户的当前密

码。SYSADMIN 或 SECURITYADMIN 角色的成员可改变密码而不用通过@OLD 提供当前密码。

- @new 是想要指定给账户的新密码。
- @loginame 是 sp_password 正要改变密码的账户用户名。

用户要改变密码有多种原因。别的用户知道了当前密码、公司策略要求定期改变密码，或许用户开始时选择了太长的密码，现在对于每次登录键入那么多字符感到厌烦等。通过调用 sp_password 并提供账户的当前密码（通过@OLD 参数）以及新的密码（通过@NEW 参数），任何用户都可改变自己账户的密码。

如果一个用户忘记了其密码，用户必须去找 SYSADMIN 或 SECURITYADMIN 角色的成员。两个组的成员都可忽略@OLD 参数，而只提供新密码（通过@NEW 参数）。例如，假定用户 Sue 想要改变其密码。为达到此目的，Sue 必须登录到 DBMS 然后按如下形式调用 sp_password 存储过程：

```
EXEC sp_password @old='OldPass', @new='newpass'
```

请注意，无权限用户必须既提供当前（老）密码又提供所希望的（新）密码。系统管理员（或 SYSADMIN 或 SECURITYADMIN 角色的任何成员）可只提供新密码（通过@NEW 参数）和用户名（通过@LOGINAME 参数）而改变另一用户账户的密码。例如，假设 Konrad 是 SYSADMIN 角色的成员，想要把 Oscar 的密码改变为 MEYER。为了改变 Oscar 的密码，Konrad 应该按如下形式调用 sp_password 存储过程：

```
EXEC sp_password @new='MEYER', @loginame='OSCAR'
```

第 25 章 通过 Internet 处理 SQL 数据库中的数据

技巧 452 使用 sp_makewebtask 创建生成 Web 页面的任务

MS-SQL Server 内建的存储过程 sp_makewebtask 使得将 SQL 的查询结果或是一组查询的结果放在 Web 页面上的 HTML 表内变得简单。为了使用 sp_makewebtask，必须事先编写想要显示其结果集的查询。由此，sp_makewebtask 并不允许 Web 站点访问者向 MS-SQL Server 提交特别的查询（读者将在技巧 465 “通过 HTML 表单提交 SQL 查询”中学习如何让站点访问者快速地编写其自己的查询）。但是，sp_makewebtask 确实允许创建具有对某些事情，如销售统计、存货目录级别、工作人员数据、顾客清单、外销商清单等的自更新报告的 Web 页面。

简短地说，如果需要重复地执行同一 SQL 查询，就可使用 sp_makewebtask 来创建一个 MS-SQL Server 将要为用户按要求或定期执行的 Web 任务。不是把查询结果显示在屏幕上，sp_makewebtask 创建的任务生成 Web 页面并将查询结果插入页面的 HTML 表内。另外，显示在 Web 页面上的 SQL 数据不必是静止的。

使用 sp_makewebtask，通过指定 MS-SQL Server Agent（MS-SQL Server 代理）定期地或是只要用户改变了数据库内的影响页面显示信息的数据就要执行的 Web 任务的查询，可创建动态的 Web 内容。

为了创建想要 MS-SQL Server 按要求（根据预先定义的时间表或是只要用户改变了报告的底层数据）执行的 Web（页面生成）任务，可调用有以下句法的 sp_makewebtask 存储过程：

```
sp_makewebtask
    [@outputfile=] '<Web page document pathname>',
    [@query=] <one or more SELECT statements>
    [,[@fixedfont=]{0|1}]
    [,[@bold=]{0|1}]
    [,[@italic=]{0|1}]
    [,[@colheaders=]{0|1}]
    [,[@lastupdated=]{0|1}]
    [,[@HTMLheader=]{1|2|3|4|5|6}]
    [,[@username=]<username>]
    [,[@dbname=]<database name>]
    [,[@templatefile=] '<pathname of HTML template>']
    [,[@webpagetitle=] '<Web page title text>']
    [,[@resultstitle=] '<Title text above HTML table(s)>']
    [,([@URL=] '<Web page URL>',[@reftext=] '<anchor text>')]{1}
    ([@table_urls=]{0|1},
    [@url_query=] '<2-column table of URL queries>')]{1}
    [,[@whentype@=]{1|2|3|4|5|6|7|8|9|10}]
    [,[@targetdate=] <date in YYYYMMDD format>]
    [,[@targettime=] <time in HHMMSS format>]
```

```
[,[@dayflags=]<day(s) of week flag>]
[,[@numunits=]<number of @UNITTYPE units in each period>]
[,[@unittype=]{1|2|3|4}]
[,[@procname=]<Web task name>]
[,[@maketask=]{0|1|2}]
[,[@rowcnt=]<maximum number of rows in results set>]
[,[@tabborder=]{0|1}]
[,[@singlerow=]{0|1}]
[,[@blobfmt=]<blob data disposition>]
[,[@nrowsperpage=]<results set rows per Web page>]
[,[@datachg=]{TABLE=<table name>[COLUMN=<column name>]
    [...TABLE=<last table name>
        [COLUMN=<last column name>]]}]
```

其中：

- **@outputfile** 指定存储过程所创建的 HTML 文档的全路径。例如，为了在 D:驱动器上的 \WEBS\SQLTIPS 文件夹中创建 INVENTORY_LIST.HTM Web 页面，应使用 **@outputfile='D:\WEBS\SQLTIPS\INVENTORY_LIST.HTM'**。
- **@query** 指定 MS-SQL Server 执行的查询或一组查询。存储过程 **sp_makewebtask** 在由 **@OUTPUTFILE** 指定的 Web 页面上的 HTML 表内显示来自每个查询的结果集。
- **@fixedfont** 为 1，如果查询结果以固定字体显示，或为 0，如果查询结果以比例字体显示。默认为 1。
- **@bold** 为 1，粗体字体显示，或为 0，如果查询结果以正常字体（非粗体）显示。默认为 0。
- **@italic** 为 1，如果查询结果以斜体显示，或为 0，如果查询结果以非斜体显示。默认为 0。
- **@colheaders** 为 1，如果查询结果集中的列名用作 HTML 表的标题，或为 0，如果显示查询结果时没有列名。默认为 1。
- **@lastupdated** 为 1，如果存储过程在有查询结果的第一个 HTML 表前插入 Last updated: <date and time>（最新更新：<日期和时间>）一行，或为 0，压缩最新生成信息的日期。默认为 1。
- **@HTMLheader** 指定当格式化@RESULTSTITLE 中的文本时使用 6 种 HTML 标题级别中的哪一种。例如，将@HTMLHEADER 设置 1，则存储过程将查询结果表上面的标题格式化为<h1>Title</h1>，设置 2，则格式化为<h2>Title</h2>，依此类推（直到 6）。
- **@username** 是执行指定给@QUERY 的查询（或几个查询）的用户名。默认值使用当前用户名（也就是创建 Web 任务的人的用户名）。只有系统管理员（SA）或是数据库拥有者（DBO）可以指定与当前用户名不同的用户名。
- **@dbname** 指定存储过程执行查询（或几个查询）的数据库名。默认时是使用当前数据库。
- **@templatefile** 是用作存储过程生成的 Web 页面的模板文件的全路径。HTML 模板包括 HTML 标记和出现在 Web 页面上除了有查询结果的 HTML 表之外的文本。存储过程将使用 SQL 查询返回的数据代替 Web 页面模板文件中的每个

<%insert_data_here%>标记。

- @webpagetitle 是存储过程在 Web 页面的标题部分的开始和结束标题标记 (<title></title>) 之间放置的文本。默认标题是 SQL Server Web Assistant。
- @resultstitle 是存储过程在 Web 页面上插入查询结果的第一个 (或许是惟一的一个) HTML 表之前作为标题而显示的文本。
- @URL 是指向另一 HTML 文档的 Web 地址 (也就是统一资源定位符或 URL)。存储过程将超级链接的 href 属性设置为作为 @URL 传递的 URL。
- @refext 是显示在 Web 页面上的查询结果的 HTML 表后的一行中的超级链接锚文本。通过将作为 @URL 和 @REFTEXT 参数传递的文本代替 @URL 和 @REFTEXT, 代替以下面的超级链接句法进行: @REFTEXT, 存储过程创建指向另一 Web 页面的超级链接。
- @table_urls 为 1, 如果存储过程在 Web 页面上插入的超级链接是由 @URL_QUERY 参数之内的查询生成的。缺省值为 0, 说明没有查询生成超级链接。如果 @TABLE_URLS 为 1, @URL_QUERY 必须有返回两列结果表的 SELECT 语句。
- @url_query 是一条可返回两列结果表 (其中包括超级链接 Web 地址 (URL) 和锚) 的 SELECT 语句。存储过程在 Web 页面上最后一个 SQL 查询结果的 HTML 表之后显示由执行查询 (在 @URL_QUERY 内) 返回的超级链接。URL 结果表内的每行的第一列包括 Web 页面的 URL, 而第二列包括超级链接的锚文本。
- @whentype 指定何时 MS-SQL Server Agent 要运行创建带查询结果的 Web 页面的 Web 任务。默认值为 1, 表明 MS-SQL Server 立即运行该任务并在执行之后立即删除该任务 (以及创建 Web 页面)。@WHENTYPE 参数的可能值有:
 - 现在创建 Web 页面。存储过程创建 MS-SQL Server 执行的 Web 任务, 然后在执行后立即删除。
 - 以后创建 Web 页面。存储过程创建以后执行的 Web 任务, 执行时间由 @TARGETDATE 和 @TARGETTIME 参数指定。在执行 Web 任务 (一次) 之后, MS-SQL Server 删除该任务 (如果忽略了 @TARGETTIME, 则 Web 任务在上午 12:00 执行)。
 - 一星期中每 n 天创建 Web 页面。存储过程创建 MS-SQL Server Agent 在一星期的某几天中运行的 Web 任务, 具体时间由 @DAYFLAGS 参数指定。MS-SQL Server Agent 在由 @TARGETDATE 的日期的 @TARGETTIME 时间开始 Web 任务, 然后在一周内每 n 天执行一次该任务 (如果忽略了 @TARGETTIME, Web 任务每 n 天在上午 12:00 开始)。
 - 每 n 分钟、小时、天或周创建 Web 页面。存储过程创建每 n 段时间 MS-SQL Server Agent 就要运行的 Web 任务。期间 (分、小时、天或周) 由 @UNITTYPE 参数指定。MS-SQL Server Agent 在由 @TARGETDATE 指定的日期的 @TARGETTIME 时间开始运行的 Web 任务, 而且每 n 段时间之后再次运行 (如果忽略了 @TARGETTIME, Web 任务在上午 12:00 开始)。
 - 根据请求创建 Web 页面。存储过程创建 MS-SQL Server 仅当用户使用 sp_runwebtask 调用该存储过程时才运行的 Web 任务。

- 现在和以后创建 Web 页面。存储过程创建 MS-SQL Server 现在执行而且 MS-SQL Server Agent 将在@WHENTYPE 为 2 时指定的时间还要运行一次的 Web 任务。
 - 现在和一星期中每 n 天创建 Web 页面。存储过程创建 MS-SQL Server 现在执行而且 MS-SQL Server Agent 在@WHENTYPE 为 3 指定的时间周期地执行的 Web 任务（而且@TARGETDATE 是不必要的）。
 - 现在以及以后周期地创建 Web 页面。存储过程创建 MS-SQL Server 现在执行而且 MS-SQL Server Agent 以后周期性地按@WHENTYPE 为 4 时指定的周期运行的 Web 任务（而且@TARGETDATE 是不必要的）。
 - 现在并根据请求创建 Web 页面。存储过程创建 MS-SQL Server 现在执行而且按@WHENTYPE 为 5 时，只要用户请求就要 MS-SQL Server Agent 运行的 Web 任务。
 - 现在以及数据变化时创建 Web 页面。存储过程创建 MS-SQL Server 现在执行而且只要用户改变了在@DATAchg 参数指明的列内的一个值改变时再次运行的 Web 任务。
- @targetdate 指定 MS-SQL Server Agent 运行 Web 任务的日期。当@WHENTYPE 为 2（以后）、3（一周几天）、4（周期的）或 6（现在和以后）时，@TARGETDATE 是必需的。@TARGETDATE 的格式是 YYYYMMDD。
 - @targettime 指定 MS-SQL Server Agent 运行 Web 任务在@TARGETDATE 指定的日期的时间。@TARGETTIME 的格式是 HHMMSS。
 - @dayflags 指定一周内的哪一天 MS-SQL Server Agent 要执行 Web 任务。如果 Web 任务要在每周内的几天都执行，可在添加日期标志时将执行各天的代码加在一起。例如，为了让 MS-SQL Server Agent 在星期一、星期二和星期五运行 Web 任务，可将@DAYFLAGS 设置为 42，即 2（星期一）+8（星期三）+32（星期五）。
 - 1 = 星期日
 - 2 = 星期一
 - 4 = 星期二
 - 8 = 星期三
 - 16 = 星期四
 - 32 = 星期五
 - 64 = 星期六

如果 Web 任务要在每周内的多天执行，可在添加日期标志时将执行各天的代码加在一起。例如，为了让 MS-SQL Server Agent 在星期一、星期二和星期五运行 Web 任务，可将@DAYFLAGS 设置为 42，即 2（星期一）+8（星期三）+32（星期五）。

- @numunits 指定周期性的 Web 任务后来要执行的分钟数、小时数、天数或是周数，也就是当@WHENTYPE 为 4（周期的）或 8（现在和周期的）时。@UNITTYPE 参数指定两次连续执行之间的时间期间）。
- @unittype 指定当@WHENTYPE 为 4（周期的）或 8（现在和周期的）时的两次 Web 任务之间的时间单位——1=小时、2=天、3=周、4=分。
- @procname 是 Web 任务的名称。如果忽略，则系统生成其名称，诸如 WEB_<YYMMDDHHMMSS><SPID>之类。

- **@maketask** 指定是否与创建生成 Web 任务的存储过程一起安排 Web 任务的时间。
 - 0——生成不加密的存储过程，但不创建 Web 任务。
 - 1——生成加密的存储过程和 Web 任务。
 - 2（默认的）——生成不加密的存储过程并加以执行。
- **@rowcnt** 指定要在 Web 页面上的 HTML 表内显示的结果集的最大行数。缺省值为 0，意味着显示结果集中的所有行。
- **@tabborder** 为 1（默认值），如果在 HTML 查询结果表周围有边框；或为 0，如果 HTML 查询结果表周围无边框。
- **@singlerow** 指定是将查询结果都显示在一页上还是显示在多页上，每页显示一行。默认值为 0，表明结果集中的所有行都出现在单个 HTML 表中。1 表明 Web 任务要为查询结果的每一行创建一个页面。相继的 HTML 页面使用追加到由 **@OUTPUTFILE** 参数指明的文件名上的一个号码来生成。例如，如果 **@OUTPUTFILE** 为 **WEB_PAGE.HTML**，而且结果集有三行，则 Web 任务要创建的 Web 页面是 **WEB_PAGE1.HTML**、**WEB_PAGE2.HTM** 和 **WEB_PAGE3.HTML**。
- **@blobfmt** 指定“集团”性的数据（也就是类型为 **IMAGE**、**NTEXT** 或 **TEXT** 的数据）是否要显示在 Web 页面的 HTML 表内，或者这些列是否应该写入外部文件并与当前 Web 页通过 URL 链接起来。请参见技巧 455 “使用 MS-SQL Server 存储过程 **sp_makewebtask** 在链接的 Web 页面上显示 **IMAGE** 和 **TEXT** 数据”，从中可获得有关处理 **IMAGE**、**NTEXT** 和 **TEXT** 数据的细节。
- **@nrowsperpage** 指定显示在每个 Web 页面上的查询结果集的行数。相继的页面使用 **NEXT** 和 **PREVIOUS** 超级链接链接在一起。

@datachg 是在发生变化之后触发 Web 任务执行的表（可选的）列名的清单。当 **@WHENTYPE** 为 10 时，**@DATACHG** 是必需的。指明 **@DATACHG** 参数就创建了对由 **@DATACHG** 参数指定的每个表的 3 个触发器（**UPDATE**、**INSERT** 和 **DELETE**）。当由 **DATACHG** 指定的表上执行了 **UPDATE**、**INSERT** 或 **DELETE** 而触发时，MS-SQL Server 将执行生成 Web 页面的存储过程。如果触发器已经存在于表上，倘若已有的触发器以 **WITH ENCRYPTION** 参数创建的，则 **sp_createwebtask** 在已有的触发器的结尾处添加其 **sp_runwebtask** 调用（如果已有的触发器是加密的，**sp_makewebtask** 将不能执行）。

例如，假设想让 MS-SQL Server 在每天的上午 12:00 创建 Web 页面（如图 25.1 所示）。如果报告所需的表位于 **SQLTips** 数据库内，可执行以下语句批处理：

```
USE SQLTips
EXEC sp_makewebtask
    @outputfile='D:\InetPub\WWWRoot\SqlTips\ProductList.htm',
    @query='SELECT p.Item_Number AS ''Item'', Description,
            Cost, Sales_Price AS ''Retail Price'',
            (SELECT COUNT(*) FROM inventory AS i
             WHERE i.item_number = p.item_number) AS
            ''On Hand''
            FROM products AS p
            ORDER BY p.item_number',
    @HTMLHeader=1,
```

```

@webpagetitle='NVBizNet Product List',
@resulttitle='Confidential Price / Product List',
@whentype=8,
@numunits=1,
@unittype=2,
@procname=web_CreateProductsPage

```

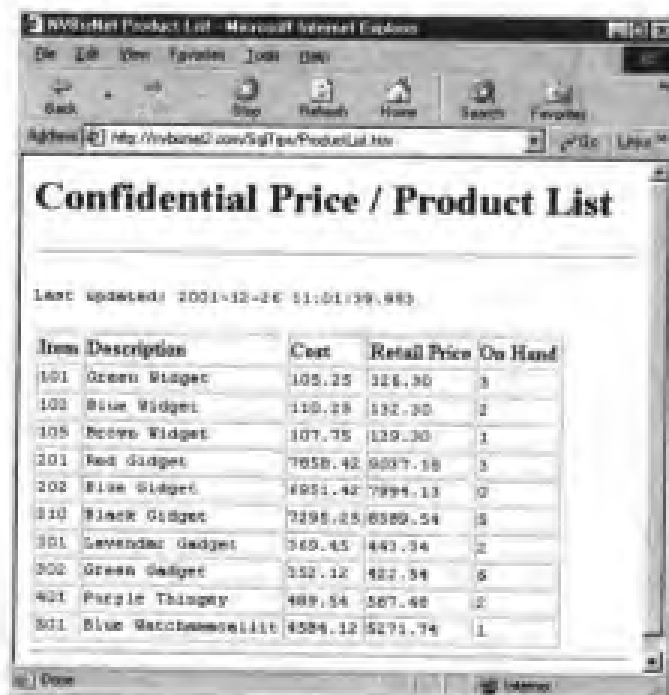


图 25.1 由 sp_makewebtask 创建的存储过程所生成的通用 Web 页面

请注意，调用 sp_makewebtask 存储过程不仅创建生成 Web 页面（如图 25.1 所示）的存储过程（在本例中是 web_CreateProductsPage），而且还有按照通过@WHENTYPE 参数指定的时间安排执行的 Web 任务。

技巧 453 为 MS-SQL Server 查询结果创建 Web 页面模板

虽然 Web 页面在 HTML 表内显示查询结果是按照所表明的那样，但大多数人很可能想要对 Web 页面的内容和布局施加更大的控制。为此，可创建一个 Web 页面模板并让存储过程将 SQL 查询结果与其他元素一起按要求插入 HTML 表内。

可为模板文件起任何喜欢的文件名，但是必须把模板保存在 MS-SQL Server 在执行存储过程根据模板创建 Web 页面时可访问的文件夹内。Web 页面模板是普通的 HTML（或 XHTML）文档，其中在想要插入查询结果的 HTML 表出现的地方有<%insert_data_here%>标记。

例如，假设想要创建类似于图 25.1 所示的 Web 页面。但是不是在白色背景下的空白文本，而是想让 Web 页面在浅黄色的背景下的蓝色文本。除此之外，还想把公司的标志放在产品清单的上部。为了控制出现在 Web 页面上由 MS-SQL Server 生成的元素，可按如下形式创建模板文件：

```

<html>
<head>

```

```

<title>NVBizNet - Product List</title>
<style type="text/css">
    h1 {color:blue; font-size:40px; font-family:verdana}
    body {color:blue; background:lightyellow;
        font-family:helvetica}
</style>
</head>
<body>
    <center>
        <h1>Confidential Price / Product list</h1>
    </center>
    <hr>
    <center><img src='images/CompanyLogo.gif'></center>
    <hr>
    <center>
        <%insert_data_here%>
    </center>
    <hr>
</body>
</html>

```

在将模板保存到 D:\INETPUB\WWWROOT\SQLTIPS\TEMPLATES\ 文件夹内名为 PRODLIST.TF 的文件中之后，可按如下所示向 sp_makewebtask 存储过程调用中添加 @TEMPLATEFILE 参数：

```

EXEC sp_makewebtask
    @outputfile='D:\InetPub\WWWRoot\SqlTips\ProductList.htm',
    @query='SELECT p.Item_Number AS ''Item'', Description,
              Cost, Sales_Price AS ''Retail Price'',
              (SELECT COUNT(*) FROM inventory AS i
               WHERE i.item_number = p.item_number) AS
              ''On Hand''
            FROM products p
            ORDER BY p.item_number',
    @templatefile=
        'D:\InetPub\WWWRoot\SQLTips\Templates\ProdList.TF',
    @whentype=8,
    @numunits=1,
    @unitttype=2,
    @procname=web_CreateProductsPage

```

模板文件 (PRODLIST.TF) 上的 .TF 扩展名是任意的，现在的扩展名代表 Template File (模板文件)。请注意，在指定 Web 页面模板时，可从 sp_makewebtask 存储过程调用中忽略所有的页面元素参数（如 @LASTUPDATED、@HTMLHEADER、@WEBPAGETITLE 和 @RESULTSTITLE）。当使用模板来指定 Web 页面的格式和（或）其他元素时，MSSQL Server 将忽略这些参数。

顺便说一下，并不局限于在每个页面上只显示单一结果集。如果想要在 Web 页面模板内显示来自多个查询的结果集，只要对每个结果集放置一个 <%insert_data_here%> 标记。例如，

如果想要将 CUSTOMERS 表内以及 EMPLOYEES 表内的数据显示为同一 Web 页面上的两个 HTML 表，可使用如下的模板：

```
<html>
<head>
  <title>NVBizNet - Customers & Employees</title>
  <style type="text/css">
    body {background:lightyellow; font-family:helvetica}
  </style>
</head>
<body>
  <center><h1>Confidential Lists</h1></center>
  <hr>
  <center><h2>Customer List</h2></center>
  <center><%insert_data_here%></center>
  <hr>
  <center><h2>Employee List</h2></center>
  <center><%insert_data_here%></center>
  <hr>
</body>
</html>
```

显示在本技巧内的特定的信息类型以及其他页面元素的杂项和格式化指令都不是重要的——可在模板内使用任何合法的 HTML 或 XHTML 元素以及文本内容。但是请注意，必须要想让 MS-SQL Server 显示来自 SQL 查询的结果表的 Web 页面处插入<%insert_data_here%>标记。在下一技巧“格式化由 MS-SQL Server 存储过程创建的 Web 页面上的查询结果表”中，读者将了解到，在<%begindetail%>和<%enddetail%>标记之间可为查询结果表的每一列插入<%insert_data_here%>标记。

在创建有多个<%insert_data_here%>标记的模板并用 CUSTS_N_EMPS.TF 之类的文件名将其保存到磁盘之后，可将多个 SELECT 语句赋予 sp_createwebtask 的@QUERY 参数。在本例中有两处<%insert_data_here%>标记，因而可将传递两个查询的@QUERY 改写如下：

```
EXEC sp_makewebtask
@outputfile='D:\InetPub\WWWRoot\SqlTips\Custs_Emps.htm',
@query='SELECT * FROM customers
SELECT * FROM employeesp'
@templatefile=
'D:\InetPub\WWWRoot\SQLTips\Templates\Custs_N_Emps.TF',
@whentype=8,
@numunits=1,
@unitttype=2,
@procname=web_CreateEmpsAndCustsPage
```

虽然为了格式化的目的只显示了两行，但第二条 SELECT 语句可开始在第一行结束的同一行上。所有需要做的只是在一个查询的最后一个字符与下一查询开始的 SELECT 关键词之间留下至少一个空格。

技巧 454 格式化由 MS-SQL Server 存储过程创建的 Web 页面上的查询结果表

除了与一个或几个查询结果表一起在页面上插入文本和图像之外,读者还了解到,Web 页面模板允许格式化文本并使页面完全按所要求的布局。事实上,可使用任何合法的 HTML 或 XHTML 文档作为 Web 页面模板。只要在想要 SQL 查询结果出现的地方插入 `<%insert_data_here%>` 标记并将更新的文档保存到新文件名中即可。接着可将 `@TEMPLATEFILE` 参数(在 `sp_createwebtask` 存储过程调用内)设置为模板文件的全路径。然后 MS-SQL Server 将用原来页面的所有元素加上一个或多个带有由 SQL 查询返回的数据的 HTML 表生成新 Web 页面。

由于 `sp_makewebtask` 提供很少的几种字型和文本格式化选项,因而为查询结果文本编写附加的格式化指令是必要的。幸运的是,可使用 `<%begindetail%>` 和 `<%enddetail%>` 标记告诉 MS-SQL Server 只要向 Web 页面模板上的每条指令格式化的 HTML 表内插入列数据。

为了格式化 Web 页面上的查询结果,必须编写在 Web 页面模板内定义 HTML 表的标记(而不是让 MS-SQL Server 创建 HTML 标记)。例如,为了格式化带有技巧 453“为 MS-SQL Server 查询结果创建 Web 页面模板”中看到的 Product/Price List 表,可以编写如下的 Web 页面模板:

```
<!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>NVBizNet - Product List</title>
  <style type="text/css">
    th {text-align:center; background:white}
    td {border-style:solid; border-width:1px 1px;
        border-color:black; padding-left:10px;
        padding-right:10px}
    thead {color:black; background:lightgreen;
           font-weight:bold; text-align:center}
    tfoot {color:white; background:blue; font-weight:bold;
           text-align:center}
    tbody {text-align:right; background:lightblue}
    h1 {color:blue; font-size:40px; font-family:verdana}
    body {color:blue; background:lightyellow;
          font-family:helvetica}
  </style>
</head>
<body>
  <center><h1>Confidential Price/Product List</h1></center>
  <hr />
  <center><img src='images/CompanyLogo.gif'></center>
  <hr />
  <center>
    <table cellpadding="0">
      <thead>
```

```

        <tr>
        <td colspan="5">Prices & Products</td>
    </tr>
</thead>
    <tfoot>
        <tr>
        <td colspan="5">
&ampcopy NVBizNet.com (702)-361-0141</td>
        </tr>
    </tfoot>
<tbody>
<tr>
<th>Item</th><th>Description</th><th>Cost</th>
<th>Retail Price</th><th>On Hand</th>
</tr>
<%begindetail%>
<tr>
        <td><%insert_data_here%></td>
        <td><%insert_data_here%></td>
        <td><%insert_data_here%></td>
        <td><%insert_data_here%></td>
        <td><%insert_data_here%></td>
    </tr>
<%enddetail%>
</tbody>
</table>
</center>
</body>
</html>

```

在样式表中的 th 和 td 规则（在本例中的开始处）设置了表标题（<th>）和表内数据（<td>）单元格的样式。除此之外，thead、tbody 和 tfoot 规则设置 HTML 表的 3 部分的外观——标题（<thead>）、主体（<tbody>）和页脚（<tfoot>）。要注意的重要事情是：

- Web 页面模板包括在开始和结束表标记（<table></table>）之间的表定义。
- 在 Web 页面模板内的开始和结束表标题标记之间（<th></th>）定义表标题单元格的文本，而不依赖 MS-SQL Server 来向结果表中添加这些元素。
- 在表定义内的表数据标记的一行两边放置<%begindetail%>和<%enddetail%>标记并在每套开始和结束数据标记（<td></td>）之内插入一个<%insert_data_here%>标记。在这种情况下，每个<%insert_data_here%>标记告诉 MS-SQL Server 插入来自查询结果集中一行内的一列的数据。

sp_createwebtask 存储过程调用仍然没有什么太大的变化：

```

EXEC sp_makewebtask
    @outputfile='D:\InetPub\WWWRoot\SqlTips\ProductList.htm',
    @query='SELECT p.Item_Number, Description, Cost,
            Sales_Price, (SELECT COUNT(*)

```

```

FROM inventory AS i
WHERE i.item_number =
      p.item_number)
FROM products p
ORDER BY p.item_number',

@templatefile=
'D:\InetPub\WWWRoot\SQLTips\Templates\FmtProdList.TF',
@whentype=8,
@numunits=1,
@unitttype=2,
@procname=web_CreateProductsPage

```

由于 HTML 表标题已经编写到 Web 页面模板中, 因而创建 Web 页面的存储过程内的 SELECT 子句不再需要提供对用户友好的标题。

要理解的最重要的事情是, 必须在 Web 页面的 HTML 表内定义如下的一行:

```

<%begindetail%>
<tr>
  <td><%insert_data_here%></td>
  <td><%insert_data_here%></td>
  <td><%insert_data_here%></td>
  <td><%insert_data_here%></td>
  <td><%insert_data_here%></td>
</tr>
<%enddetail%>

```

MS-SQL Server 用来自结果集当前行中的列数据代替<%insert_data_here%>标记, 然后对下一行再次使用表定义, 依此类推。在本例中, SQL 查询的 SELECT 子句有 5 列, 因而表定义也必须有 5 套开始和结束表数据标记 (<td></td>), 才能创建有 5 个数据单元格的行。如果 SELECT 子句有 10 列, 在 Web 页面模板上的 HTML 表内定义的表数据行需要 10 个单元格 (也就是说, 要有 10 套开始和结束数据标记 [<td></td>])。

技巧 455 使用 sp_makewebtask 在链接的 Web 页面上显示 IMAGE 和 TEXT 数据

MS-SQL Server 允许在数据类型为 TEXT 列中保存多于 8,000 个字符的字符串, 而在数据类型为 IMAGE 的列中可保存图像文件(单个 TEXT 和 IMAGE 类型的列可保存多达 2GB 的数据)。虽然可以使用在技巧 452~技巧 454 所学习的内容来执行查询并让 MS-SQL Server 在 HTML 表的 TEXT 类型列的单元格内显示其内容, 但通常最好不要这样做。用于显示 TEXT 数据的列内的数据量将会使 HTML 表内其余数据项变小, 因而当用表式显示相关信息时, 使表变得不能用。

当在 SQL 表内保存图像时, DBMS 将图像保存为二进制的字符串 (也就是一串 1 和 0)。MS-SQL Server 将其留给从 IMAGE 列提取数据的应用程序来翻译 1 和 0 的串并重新构建二进制串所代表的图像。因此, 最好不要显示 HTML 表内单元格中的 IMAGE 列的内容。DBMS 会将 IMAGE 数据作为二进制串放入表的单元格内而不是人们想要看到的图像。

内建的存储过程 sp_makewebtask 允许使用 @BLOBFMT 参数来指定 DBMS 对 TEXT 和 IMAGE 列内的数据做什么。既可让 DBMS 在 HTML 表的单元格内显示 TEXT 和 IMAGE 数

据 (正如所讨论过的, 这不是所希望的), 也可以让 DBMS 将数据写入外部文件并在 HTML 表上放一个超级链接。在查询结果的 HTML 表中放置一个超级链接是个好的解决办法, 因为这允许以表的形式查看查询结果, 而且, 如果想要读取大的 (也就是长的) 文本项或查看图像, 只需要单击其 HTML 表内的超级链接即可。

为了将 TEXT 或 IMAGE 类型的列写入外部文件, 可用以下句法使用 `p_makewebtask` 存储过程的 `@BLOBFMT` 参数:

```
@BLOBFMT='%<TEXT/IMAGE column number>%
        file=<output filename>
        [tplt=<template filename>]
        URL=<Web address of output file>
        [...%<last TEXT/IMAGE column number>%
        file=<last output filename>
        [tplt=<last template filename>]
        URL=<Web address of last output file>]'
```

请注意, 可以处理多个 TEXT 和 (或) IMAGE 数据列。只要在 `@BLOBFMT` 参数内重复 `%<column #>%[<template file>]<Web address>` 样式, 对每个 TEXT 和 IMAGE 列重复一次 (每个外部文件可通过 SQL 查询结果的 HTML 表内的超级链接加以访问)。

例如, 假设 `sp_makewebtask` 的 `@QUERY` 参数有如下的 SELECT 语句, 可返回 1 号列中的 TEXT 数据以及 6 号列中的 IMAGE 数据:

```
SELECT pr_info AS 'Publisher Name', pub_name, city, state,
       country, logo, 'Company Logo'
FROM pub_info AS pub_info, publishers
where pub_info.pub_id = publishers.pub_id
```

给定前面的查询, 本例中的 `@BLOBFMT` 参数可设置如下:

```
@blobfmt=
    '%1% file=D:\InetPub\WWWRoot\SqlTips\Temp\PR_Text.htm
    URL=http://NVBizNet2.com/SQLTips/Temp/PR_Text.htm
    %6% file=D:\InetPub\WWWRoot\SqlTips\images\publogo.gif
    URL=http://NVBizNet2.com/SQLTips/images/publogo.GIF'
```

这样一来, `sp_createwebtask` 存储过程调用的全部文本有点像下面的样子, 可生成如图 25.2 所示的 Web 页面。

```
USE pubs
EXEC sp_makewebtask
    @outputfile='D:\InetPub\WWWRoot\SqlTips\Publishers.htm',
    @query='SELECT pr_info AS ''Publisher Name'', pub_name,
                city, state, country, logo,
                ''Company Logo''
            FROM pub_info AS pub_info, publishers
            where pub_info.pub_id = publishers.pub_id',
    @blobfmt='%1%
file=D:\InetPub\WWWRoot\SqlTips\Temp\PR_Text.htm
URL=http://NVBizNet2.com/SQLTips/Temp/PR_Text.htm
%6%
file=D:\InetPub\WWWRoot\SqlTips\images\publogo.gif'
```



```
URL=http://NVBizNet2.com/SQLTips/images/publogo.GIF',
@whentype=9,
@procname=web_CreatePublishersPage
```



图 25.2 由存储过程 web_CreatePublisherPage 所生成的其中有指向 TEXT 和 IMAGE 数据的通用 Web 页面

存储过程使用了查询结果的 IMAGE 或 TEXT 列后面的列内容作为指向外部文件（TEXT 和 IMAGE 数据写入其中）的超级链接的锚文本。在本例中，列 2（PUB_NAME）返回的值作为指向外部文件（其中有由列 1 [PR_INFO] 返回的 TEXT 数据）的超级链接的锚文本字符串。类似地，列 7（实际字符串 'Company Logo'）中返回的值作为指向保存有由列 6（LOGO）返回的 IMAGE 数据的外部文件的超级链接的锚文本字符串。

这样一来，正如前面的图 25.1 所示，存储过程并不在 HTML 表的列内显示 TEXT 和 IMAGE 内容。而是存储过程将 BLOB 内容写入磁盘文件并将一个指向该内容的超级链接放于 HTML 表内。请注意，在本例中，查询返回 7 列，只有 5 列显示在 HTML 表内，因为两列（第 1 列和第 6 列）已写入磁盘文件。

虽然本例没有使用模板文件，但可为主 Web 页面指定一个 HTML 模板（也就是带查询结果的 HTML 表的 Web 页面），再为存储过程写入 TEXT 列的数据的 Web 页面指定另一模板（不能对 IMAGE 列使用模板文件，因为想让存储过程将表的 IMAGE 列内的二进制串完全按原样写入磁盘上的图形文件）。

注意：对于 TEXT 数据，可使用带有 Web 浏览器作为 Web 页面加以识别的扩展名（也就是 .htm、.html、.asp、.php 等）的文件名。为了让 .php 和 .asp 也好用，Web 服务器必须装有 Asp 所用的 Active Server Pages 脚本引擎或 PHP 所用的 PHP 脚本引擎。对于 IMAGE 数据，可使用能够响应图像编码格式的文件扩展名（通常为 .GIF、.JPG、.TIF 等）。通过使用 TEXT 数据的 Web 页面扩展和 IMAGE 数据的图形文件扩展，可指令 Web 浏览器将 TEXT 数据的文件显示为 Web 页面

（其中有许多字符的长字符串），并将 IMAGE 文件内的二进制串翻译为图形图像。

技巧 456 使用内建的存储过程启动或删除 Web 任务

MS-SQL Server 有 3 个可用于管理 Web 任务的内建的存储过程：`sp_createwebtask`、`sp_runwebtask` 和 `sp_dropwebtask`。虽然 `sp_createwebtask` 允许创建新的 Web 任务，但 `sp_runwebtask` 允许执行已有的任务——既可按时间安排执行也可以不是。最后，当不再想要执行特定的 Web 任务时，可使用内建的存储过程 `sp_dropwebtask` 从 DBMS 删除 Web 任务。

读者在技巧 452~技巧 455 中学习了如何使用 `sp_createwebtask`。简短地说，`sp_createwebtask` 允许创建特殊类型的存储过程，可执行 DBMS 将其结果集写到 Web 页面上的 HTML 表中的一个或多个查询。在示例中可看出来，但并未在前 4 个技巧中指出的一件事是，应该总是使用 `sp_createwebtask` 的 `@PROCNAME` 参数来命名所创建的每个 Web 任务。确实如此，可以忽略掉该参数并让系统生成名称。但是，系统生成的名称形式是 `web_<YYMMDDHHMMSS><SPID>`，这个名称不容易记住。除此之外，系统生成的名称，如 `web_20011227042606552682` 并不能说明有关存储过程与什么样的 Web 任务有关的任何信息。因此，当手工开始一项 Web 任务时或者为了编辑或删除，必须搜索书面文档或必须编辑存储过程以便看出哪个是想要运行、改变或是删除的。通过使用有描述性的名称（如 `web_CreateProductList`）就可简单地查看一个数据库内定义的 Web 任务的清单，就可找到想要工作其上的 Web 任务。

在创建 Web 任务时，可指定何时以及多长时间让 MS-SQL Server Agent 来执行该任务。但是，有时可能想要在两次安排的执行中间手工启动任务。例如，假设有一个安排好的任务，用公司每个星期天中午 12:00 时的存货目录来重新创建 Web 页面。如果接受了大量的产品发货在星期二到达，此时最好还是立即更新存货目录，而不要等到周末去。

内建的存储过程允许立即启动 Web 任务（即使在将来有一个安排好时间的执行），其句法如下：

```
sp_runwebtask [[@procname=]'<Web task process name>']  
               [,[@outputfile=]'<pathname of output file>']
```

其中：

- `@procname` 是想要执行的 Web 任务的名称。
- `@outputfile` 是 Web 任务创建的 Web 页面的名称。

当启动一个 Web 任务时，必须处于与该 Web 任务创建时处于同一数据库中。为了启动该任务，可调用 `sp_runwebtask` 并提供任务名或是该任务所要创建的 Web 页面的全路径。例如，假设在—项名为 `web_CreatePublishersPage` 的 Web 任务，可在 `D:\INETPUB\WWWROOT\SQLTIPS\` 文件夹中创建 `PUBLISHERS.HTM` Web 页面。此时可使用以下两条语句之一来调用 `sp_runwebtask` 立即运行 Web 任务：

```
EXEC sp_runwebtask @procname='web_CreatePublishersPage'  
EXEC sp_runwebtask @outputfile=  
                    'D:\InetPub\WWWRoot\SqlTips\Publishers.htm'
```

如果想要删除 Web 任务，可调用 `sp_dropwebtask` 从 DBMS 删除 Web 任务及其有关的存储过程。为了调用 `sp_dropwebtask`，可使用与调用 `sp_runwebtask` 同样的句法：

```
sp_dropwebtask [[@procname=]'<Web task process name>']
```

```
[,[@outputfile=] '<pathname of output file>']
```

与 `sp_runwebtask` 的情况一样, 既可提供 Web 任务名, 也可提供其输出文件的全路径。由此, 为了删除名为 `web_CreatePublishersPage` 的 Web 任务 (在 `D:\INETPUB\WWWROOT\SQLTIPS\` 文件夹中创建 `PUBLISHERS.HTM` 页面), 首先执行 USE 语句移动到创建 Web 任务时的 DBMS 上, 然后执行下列语句之一来删除 Web 任务及其存储过程。

```
EXEC sp_dropwebtask @procname='web_CreatePublishersPage'  
EXEC sp_dropwebtask @outputfile=  
    'D:\inetpub\WWWroot\SqlTips\Publishers.htm'
```

技巧 457 使用 MS-SQL Server Web Assistant Wizard 创建执行存储过程的 Web 任务

在技巧 452 “使用 `sp_makewebtask` 创建生成 Web 页面的任务”中, 读者学习了如何使用内建的存储过程 `sp_makewebtask` 来创建在 Web 页面上显示 SQL 数据库数据的 Web 任务。虽然 `sp_makewebtask` 易于使用, 但可用来格式化输出数据以及安排任务时间的参数的数目有点吓人。在熟悉了可用的所有选项之前, 可以考虑使用 MS-SQL Server 的 Web Assistant Wizard (Web 助手向导) 来创建 Web 任务。

正如读者在本技巧中所学习的, Web Assistant Wizard 指导用户完成创建 Web 任务的全部过程。通过把有关选项在各种对话框中组合在一起, 向导使得选项的清单易于处理。除此之外, Web Assistant Wizard 还帮助人们形成想让 Web 任务执行的查询, 允许选择想要在 Web 页面上看到数据的表列 (从图形显示中)。根据在对话框中的输入和选择 (在选择了想要显示的 SQL 表列之后显示的), 向导生成选择要包括在 HTML 表 (由 Web 任务在创建时插入 Web 页面的) 中的数据行的查询。

Web Assistant Wizard 位于 MS-SQL Server Enterprise Manager 内。为了启动向导并创建执行存储过程的 Web 任务, 可执行下面的步骤:

1. 单击 Start (开始) 按钮。Windows 显示 Start 菜单。
2. 将鼠标移动 Start 菜单的 Programs 上, 选择 Microsoft SQL Server 组, 然后单击 Enterprise Manager。Windows 将启动 SQL Server Enterprise Manager。
3. 单击 Microsoft SQL Servers 图标左边的加号 (+) 显示 SQL Server Group。然后单击 SQL Server Group 图标左边的加号 (+), 显示在网络上可用的 MS-SQL Servers 的清单。
4. 为了显示包括想要在其上创建 Web 任务的数据库的 MS-SQL Server 上的资源, 可单击 MS-SQL Server 名称左边的加号 (+)。例如, 如果想要处理名为 NVBizNet2 的 MS-SQL Server, 可单击 NVBizNet2 左边的加号 (+)。Enterprise Manager 将显示代表在其上可用的资源的文件夹清单。
5. 单击 Databases 文件夹。然后选择 Tools 菜单上的 Wizards 选项。Enterprise Manager 将选择 Select Wizard 对话框。
6. 单击 Management 左边的加号 (+), 显示 Management Wizards (管理向导) 清单。接着在 Management Wizards 清单中单击 Web Assistant Wizard, 然后单击对话框底部的 OK 按钮。Web Assistant Wizard 将显示其 Welcome (欢迎) 屏幕。
7. 单击 Next 按钮。Web Assistant Wizard 将显示 Select Database (选择数据库) 对话框。
8. 单击 Database name 域右边的下拉按钮并选择想要创建 Web 任务的数据库。对本例来

说，从选择清单中选择 SQLTips，然后单击 Next 按钮。Web Assistant Wizard 将显示 Start a New Web Assistant Job（开始一新的 Web 帮助工作）对话框，如图 25.3 所示。

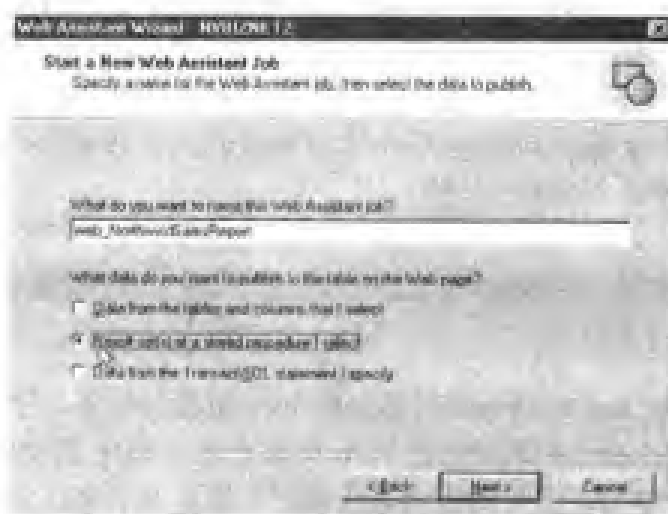


图 25.3 Web Assistant Wizard 的 Start a New Web Assistant Job 对话框

9. 向 What do you want to name this Web Assistant job?（怎样命名这个 Web 帮助工作）域中输入 Web 任务的名称。对本例来说，可输入 web_NorthwindSalesReport。

10. 在 Web 任务名域之下的单选按钮列中单击 Result set(s) of a stored procedure I select 左边的单选按钮。请注意，通过单击清单中的第一个单选按钮还可使用 Web Assistant Wizard 创建公布查询结果的 Web 任务，而通过单击第 3 个单选按钮可创建来自执行 Transact-SQL 语句所生成的结果集。单击 Next 按钮。Web Assistant Wizard 将显示 Select Stored Procedure（选择存储过程）对话框。

11. 在 Select Stored Procedure 对话框中部的有存储过程的列表框内，单击想要 Web 任务执行的存储过程。对本例来说，单击 usp_ShowNorthwindSales，然后单击 Next 按钮。Web Assistant Wizard 将显示 Schedule the Web Assistant Job（安排 Web 帮助工作的时间）对话框，如图 25.4 所示。

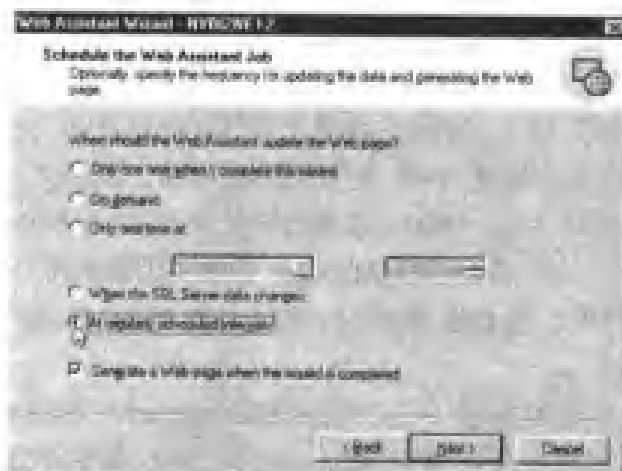


图 25.4 Web Assistant Wizard 的 Schedule the Web Assistant Job 对话框

12. 决定何时想让 Web 任务运行以及多长时间再重复运行。接着，单击选项旁边的单选按钮。对本例来说，单击 At regularly scheduled intervals（按通常安排的时间间隔）单选按钮，然

后单击 Next 按钮。Web Assistant Wizard 将显示 Schedule the Update Interval（安排更新间隔）对话框，如图 25.5 所示。

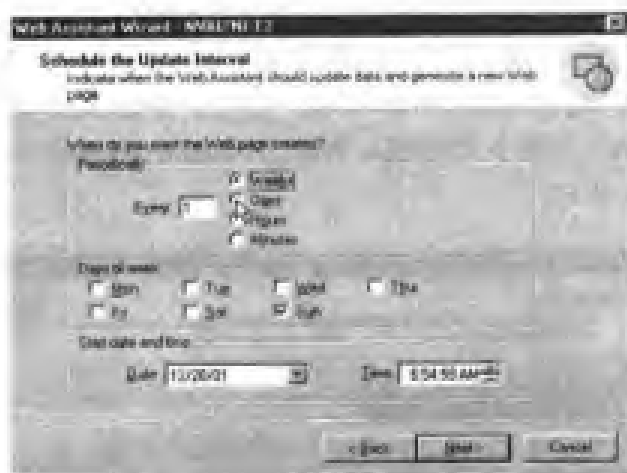


图 25.5 Web Assistant Wizard 的 Schedule the Update Interval 对话框

13. 使用对话框中的复选框和单选按钮来选择想让 Web 任务运行的日期、时间和（或）每周内的星期几。对本例来说，让 Web 任务每月更新 Northwind 的销售报告，可单击 Days（天数）单选按钮，在对话框的顶部的 Periodically（周期性的）部分内的 Every 域内输入 30。然后单击 Next 按钮。Web Assistant Wizard 将显示 Publish the Web Page（发布 Web 页面）对话框。

14. 在 Publish the Web Page 对话框内的 File name 域内输入 MS-SQL Server 发布 Web 页面的路径名。如果该文件夹对 MS-SQL Server 是可访问的，请输入有 SQL 数据显示其上的 Web 页面所在 Web 站点上一个文件夹内的 Web 页面的路径名。请注意，如果输入了不是指向 Web 站点内的一个文件夹的位置的路径，必须把 Web 任务所创建的 Web 页面复制到该站点的一个文件夹之内，然后站点访问才能看到所生成的 Web 页面。对于本例来说，假定 MS-SQL Server 有对 Web 站点文件夹的写入权限。因此，请在 File name 域内输入 D:\inetpub\WWWroot\SQLTips\NorthwindSalesReport.htm。接着，单击 Next 按钮。Web Assistant Wizard 将显示 Format the Web Page（格式化 Web 页面）对话框，如图 25.6 所示。

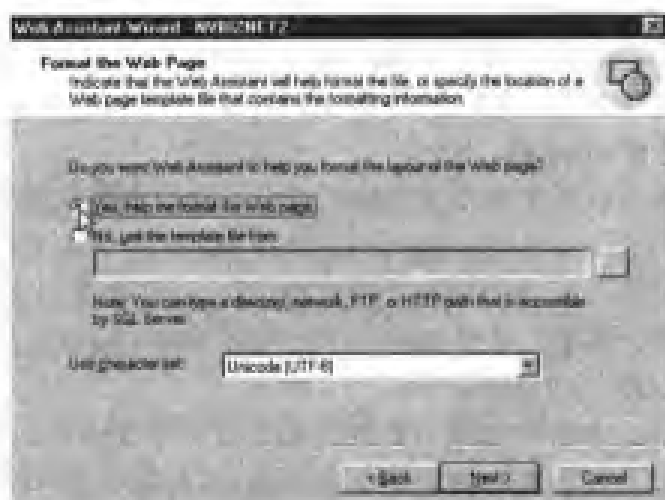


图 25.6 Web Assistant Wizard 的 Format the Web Page 对话框

15. 如果想让 Web 任务使用 Web 页面模板, 可单击 No, use the template file from 左边的单选钮, 然后输入想让 Web 任务使用的 Web 页面模板的路径名 (在技巧 453 “为 MS-SQL Server 查询结果创建 Web 页面模板” 和技巧 454 “格式化由 MS-SQL Server 存储过程创建的 Web 页面上的查询结果表” 中, 读者已经学习了如何创建 Web 页面模板)。对本例来说, 假定没有 Web 页面模板并想让 Web 任务格式化 Web 页面。因而, 单击 Yes, help me format the Web page (是, 请帮助我格式化 Web 页面) 单选钮。Web Assistant Wizard 将显示 Specify Titles (指定标题) 对话框, 如图 25.7 所示。

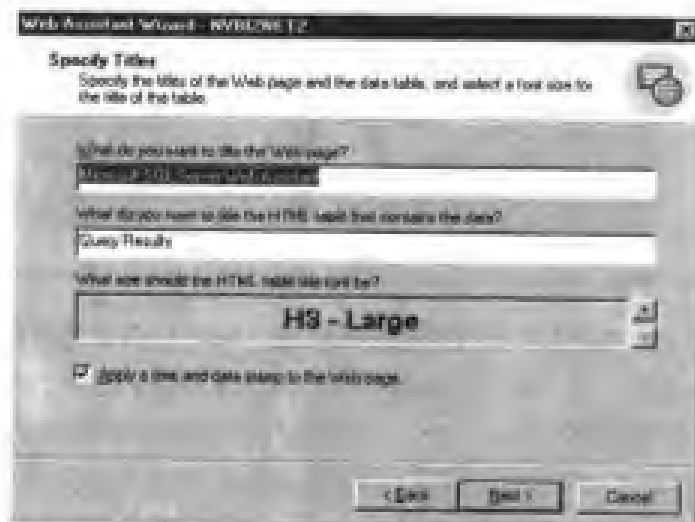


图 25.7 Web Assistant Wizard 的 Specify Titles 对话框

16. 在 “What do you want to title the Web page?” 域中键入想让 Web 任务在 Web 页面标题部分的开始和结束标记 (<title></title>) 之间插入的标题文本 (Web 浏览器在其标题栏上显示 Web 页面标题, 而不是在 Web 页面本身上显示)。对本例来说, 键入 Northwind Cumulative Sales Report。

17. 在 “What do you want to title the HTML table that contains the data?” 域内键入想要出现在 Web 页面的有查询结果的 HTML 表上面的标题文本。对本例来说, 键入 Northwind Cumulative Annual Sales Figures。

18. 使用 What size should the HTML table title font be? 右边的加号 (+) 和减号 (-) 按钮设置标题文本的字号。对本例来说, 假定想使用 HTML 的一级标题标记格式化标题, 因而单击加号 (+) 按钮直到看到 H1-Largest 为止。

19. 决定是否想让 Web 任务在 Web 页面插入有关 Web 页面最近更新的时间和日期的一行。如果不, 可单击 Apply a time and date stamp to the Web page (对 Web 页面使用时间和日期邮戳) 复选框, 使其选择标记消失。对本例来说, 假定想让 Web 任务显示最近更新的日期和时间, 因而对该复选框可不改变。接着, 单击 Next 按钮。Web Assistant Wizard 将显示 Format a Table (格式化表) 对话框, 如图 25.8 所示。

20. 确定是否想让 Web 任务使用来自查询的 SELECT 子句中的列名作为 HTML 表内第一行上标题以及确定无标题的数据看起来是什么样子的。那么可对适当的单选钮和复选框作出选择。对本例来说, 可接受默认情况, 让 Web 任务使用 SELECT 子句列名作为 HTML 表标题, 以固定宽度间距、非粗体、非斜体字体写入表数据, 并围绕 HTML 表及其单元格画出边框。

接着单击 Next 按钮。Web Assistant Wizard 将显示 Add Hyperlinks to Web Page (向 Web 页面添加超级链接) 对话框, 如图 25.9 所示。

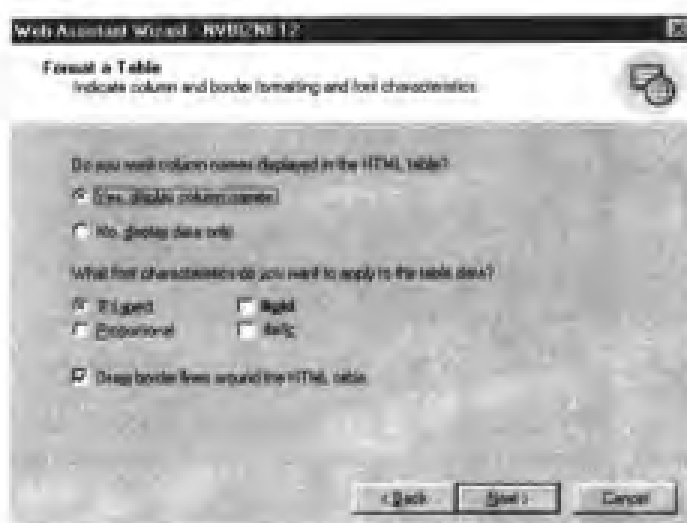


图 25.8 Web Assistant Wizard 的 Format a Table 对话框

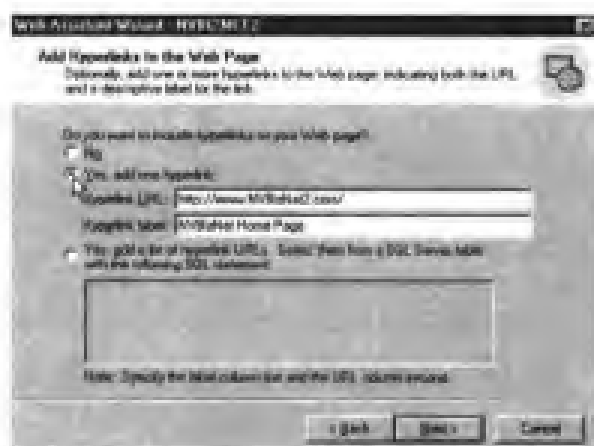


图 25.9 Web Assistant Wizard 的 Add Hyperlinks to the Web Page 对话框

21. 应该总是提供至少一个超级链接, 这样访问者可用来导航到站点的主页上, 或在站点的层次上导航到下一页或是导航到站点地图或菜单上。对本例来说, 让 Web 任务在有 SQL 数据的 HTML 表后的一行上插入一个指向站点主页 (www.NVBizNet2.com) 的超级链接。因而单击 Yes, add one hyperlink 单选钮, 在 Hyperlink URL 域内键入 HTTP://www.NVBizNet2.com/ 并在 Hyperlink label 域内键入 NVBizNet Home Page。单击 Next 按钮。Web Assistant Wizard 将显示 Limit Rows (限制行) 对话框, 如图 25.10 所示。

注意: 可以让 Web 任务在 Web 页面底部显示超级链接的清单 (而不是显示一个超级链接)。为达此目的, 必须让 SQL 表在一列中有超级链接, 而在另一列中有相关的锚文本。接着, Add Hyperlinks to the Web Page (向 Web 页面添加超级链接) 对话框底部的文本框中, 键入一条对表内每个超级链接返回两列 (一列是 <Web address> 另一列是 <anchor text>) 的 SELECT 语句。

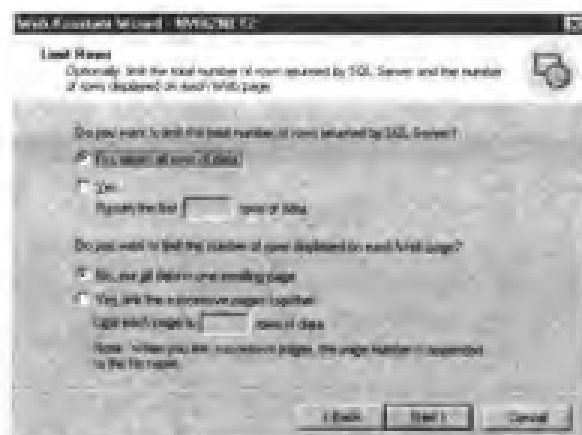


图 25.10 Web Assistant Wizard 的 Limit Rows 对话框

22. 如果想要限制 Web 任务显示的结果集行的数目或只在 HTML 表内显示某些数目或行，可单击适当的单选按钮并键入行数。对本例来说，接受 Web 默认值，在单一 HTML 表内显示查询结果表的所有行。接着单击 Next 按钮。

在完成步骤 22 后，Web Assistant Wizard 将显示其最后屏幕，其中有文本框列出了已经选择的选项。复核一下这些选项并在必要时使用 Back 按钮，返回向导对话框加以改正。

当所有都没有问题之后，单击 Finish 按钮。Web Assistant Wizard 将生成所选数据库内的 Web 任务并显示 Web Assistant successfully completed the task. (Web Assistant 成功完成任务) 消息框。

技巧 458 下载并安装 PHP

PHP（原来代表 Personal Home Page 工具）与 ASP 一样，允许运行服务器端的嵌入 Web 页面的脚本。这些嵌入的脚本可以访问数据库数据和其他对 Web 服务器可用的资源来快速地建立 Web 页面。使用 PHP 是免费的，这一点也像 ASP。但是与 ASP 不同，PHP 并不是 Web 服务器的标准附件。为了使用 PHP，正如本技巧所述，必须通过 Internet 下载。

当站点访问者请求 PHP Web 页面时，也就是通常具有与 PHP 处理器关联的.php 扩展名的 Web 文档时，Web 服务器向 PHP 处理器发送请求。PHP 处理器提取文档并一行一行地执行处理器在开始 PHP（<?php 或 <?）和结束 PHP 脚本标记间括着的 PHP 语句。PHP 处理器将语句生成的输出以及所有文本和开始和结束 PHP 标记外部的 HTML 写入 Web 服务器内存中的虚拟 Web 页面。当 PHP 处理器发出指令这样做时，Web 服务器将这个（内存中的）Web 页面发送给请求 PHP 文档的站点访问者。

简短地说，每当站点访问者请求 Web 站点上带有.php 扩展名的文档时，PHP 处理器都创建 Web 页面。因此，可使用 PHP 将静态 Web 页面集合的站点变为基于 Web 服务器的有 Web 页面用户界面的数据库应用程序。

虽然客户端脚本语言（如 JavaScript）运行在 Web 浏览器内，但 PHP 却是服务器端的脚本语言，这就意味着 Web 服务器上的程序（而且不是 Web 浏览器内的代码）执行嵌入 PHP Web 页面的 PHP 语句。用技术术语来说，PHP 是跨平台的标记语言，是嵌入服务器端的脚本语言，这就意味着：

- 可以在运行各种操作系统，如 Linux、Mac OS、RISC OS、UNIX 和 Windows 的 Web

服务器上使用同一 PHP 脚本。

- 可把 PHP 语句单独地或与向 Web 浏览器描述 Web 页面的 HTML、XHTML 或 XML 标记一起嵌入 Web 页面文件。
- 当站点访问者请求有.php 扩展名的 Web 页面时, Web 服务器将请求发送给 PHP 处理器。PHP 处理器找出所请求的文件并执行嵌入 Web 文档的脚本。PHP 处理器在服务器的内存中构建虚拟 Web 页面时, 用其输出(可能包括来自 SQL 查询的结果集内返回的数据)代替 PHP 语句。然后 Web 服务器将由运行在 Web 服务器上的 PHP 处理器构建的 Web 页面发送给站点访问者。

使用 PHP 的美妙之处在于, Web 浏览器绝不会看到嵌入 Web 页面的 PHP 代码。当编写 PHP 脚本时, 不必关心访问者的 Web 浏览器是否支持 PHP。为了创建有 PHP 功能的 Web 页面, 只要把想要执行的 PHP 脚本嵌入 Web 页面并把文档保存到带有与 PHP 处理器的执行相关联的扩展名(如.php)的文件中。当站点访问者请求带.php 扩展名的 Web 页面时, Web 服务器知道应将请求发送到 PHP 处理器然后再将 PHP 处理生成的 Web 页面发送给请求该页面的访问者。

必须下载并安装一种 PHP 处理器, 然后才能执行 PHP 脚本。如果使用的是 Linux 或 UNIX 机器, 不仅必须提取 PHP 处理器的源代码, 而且还必须使用 ANSI C 编译器(如 gcc 或 g++) 将其编译。可从 <http://www.php.net> 站点上提取最新版本的用于 UNIX/Linux 的 PHP。一定既要下载要编译的 PHP 源代码也下载 PHP 文档, 后者将对安装过程给予指导并帮助选择必要的配置选项。

在运行 IIS 的 Windows NT 上配置 PHP 比为运行在 Linux 或 UNIX 机器上的 Apache Web 服务器编译并配置 PHP 要简单一些。进入 PHP 的 Web 站点 <http://www.php.net> 并单击 Downloads (下载) 超链接。在下载 Web 页面的 Win32 Binaries 部分, 单击指向下载 PHP 归档文件的超级链接, 然后单击指向下载 PHP 安装程序的超级链接(在本书写作期间, 可分别下载 PHP 4.0.6 Zip 和 PHP 4.0.6 Installer)。当提示时, 将两个文件保存在 Web 服务器的同一文件夹(如 C:\PHP)中。请记住, PHP 是服务器端的脚本语言, 因而, 安装和运行语言处理器都是在 Web 服务器上进行的。

接着将 PHP 压缩文件解压缩, 然后执行 PHP 的 InstallShield 安装程序。安装程序将提示输入存放解压缩文件的文件夹的路径名以及想要添加 PHP 支持的 IIS(或 Personal Web Server) 的版本。在完成以上工作之后, 安装程序将要求重新引导 Windows NT/2000/XP 服务器, 从而完成安装过程。如果在安装中有什么问题或是想要执行手工安装, 可打印并查看 PHP 文件夹中的 install.txt 的内容。

技巧 459 建立数据源(DSN)与 SQL DBMS 的连接

为了处理由 SQL DBMS 管理的数据库中的数据, 应用程序(如在 ASP 或 PHP Web 页面上嵌入的脚本)必须首先登录。在创建了 DSN 之后, 登录过程是简单的。由于 DSN 为 ODBC 驱动程序提供了所有的会话信息, 脚本只需要指定与 DBMS 连接所使用的 DSN 并提供合法的用户名/密码对。简短地说, 通过 DSN 登录不比通过 SQL Query Analyzer 的登录屏幕登录到 MS-SQL Server 更困难。

例如, 在嵌入 PHP Web 页面时(这就是带有.php 扩展名的 Web 页面), 下面的 JavaScript

将使用在前一技巧中创建的 DSN 与名为 NVBizNet2 的 MS-SQL Server 连接：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<html>
<head>
    function open_DSN_connection()
    (
        $conn = odbc_connect("SQLTips", "Maggy", "Evans");
        return $conn;
    )
</head>
<body>
<?php
//Connect Call the function to connect with the DBMS
    $conn = open_DSN_Connection();
//Check the connection status
    if(!$conn)
        echo "Failed to connect!";
    else
    (
//Code that works with the data within the DBMS goes here
        echo "DSN Connection through SQLTips successful!";
    )
?>
</body>
</html>
```

在本例中，PHP 脚本使用 `odbc_connect()` 函数建立与 DBMS 的连接。函数的 3 个参数是 SQLTips（DSN 的名称）、Maggy（用户名）和 Evans（密码）。如果连接成功，`odbc_connect()` 函数返回指向 \$CONN 变量的连接句柄。虽然本例只检查连接句柄内的值，然后显示连接状态，脚本将使用连接句柄（\$CONN）向 DBMS 发送查询和其他命令并提取查询结果——正如读者将在技巧 463~技巧 465 中所学习的那样。

虽然使用 `odbc_connect()` 函数建立 PHP Web 页面上的脚本和 SQL DBMS 之间的连接，但可使用 ADO 的 Connection 对象在嵌入 ASP Web 页面的 VBScript 内做同样的事情。例如，下面的脚本将使用与前例中同一 SQLTips DSN 与 SQL DBMS 连接。但是，正如图 25.11 所示，这个脚本比前面的 JavaScript 示例提供了稍微详细一些的连接情况：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<html>
<body>
<%
'Function which establishes a connection with a DBMS
'through the DSN "SQLTips" when called.
Sub open_DSN_connection (byref connObjDSN)
    CONST dsncConnection = "DSN=SQLTips;UID=Maggy;pwd=Evans;"
'Create the ADO Connection object
    Set connObjDSN = server.createobject("adodb.connection")
'Place the connection string into the ConnectionString
```

```

'property with the ADO Connection object and then try to
'establish a connection with the DBMS.
    With connObjDSN
        .ConnectionString = dsNConnection
        .open
    End With
End Sub
DIM connObjDSN
call open_DSN_Connection (connObjDSN)
'After calling the function that opens the connection,
'display on the PHP Web page the connection details
'available from ADO Connection object properties.
    With connObjDSN
        Response.write _
            '<b>Attributes</b> = ' & .Attributes & '<br>' & _
            '<b>ADO Provider</b> = ' & .Provider & '<br>' & _
            '<b>Command Timeout</b> = ' & _
            & .CommandTimeout & '<br>' & _
            '<b>Default Database</b> = ' & _
            & .DefaultDatabase & '<br>' & _
            '<b>Connection String</b> = ' & _
            & .ConnectionString & '<br>' & _
            '<b>Connection TimeOut</b> = ' & _
            & .ConnectionTimeout & '<br>' & _
            '<b>Provider</b> = ' & .Provider & '<br>' & _
            '<b>CursorLocation</b> = ' & _
            & .CursorLocation & '<br>' & _
            '<b>Isolation Level</b> = ' & _
            & .IsolationLevel & '<br>' & _
            '<b>State</b> = ' & .State & '<br>' & _
            '<b>Version</b> = ' & .Version & '<br>' & _
    End With
%>
</body>
</html>'

```

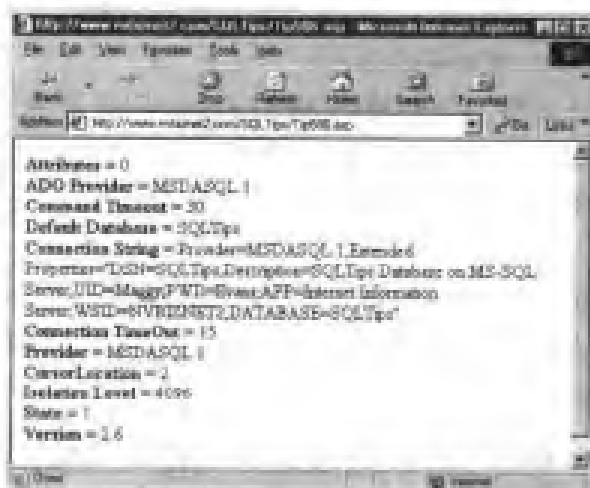


图 25.11 在 ADO Connection 对象属性内保存的有关打开的 DSN 连接的信息

请注意，ADO 对象（如本例中使用的 ADO 的 Connect 对象[connObjDSN]）有“方法”和“属性”。所谓方法是对象可做的事情或行动。例如 ADO Connection 对象有下面的方法：

- Cancel——指示 DBMS 取消异步“打开”的最后命令或通过 Connection 对象发送到 DBMS 的 SQL 语句的执行。
- Open——打开与 DBMS 的连接。
- Close——关闭对象已打开的与 DBMS 的连接。
- BeginTrans——开始 DBMS 上的事务处理。
- CommitTrans——提交（COMMIT），也就是使最近的 BeginTrans 方法调用以来所完成的工作永久化。
- RollBackTrans——取消由最近的 BeginTrans 方法调用以来所完成的工作。
- Execute——提交不期望向 DBMS 返回任何查询结果行的供执行的语句。

另一方面，一个对象的属性告诉有关对象的一些事情或者保存着由一个对象方法放入对象内的某一值或脚本向方法传递的某个参数值。例如，ADO Connection 对象有下列属性：

- Attributes——指定在执行了 COMMIT 或 ROLLBACK 语句之后是否开始新的事务处理。
- CommandTimeout——指明要等待命令执行多长时间（以秒计）。缺省值为 10 秒。
- DefaultDatabase——当 Open 方法打开与 DBMS 的连接时，让用户指定连接的初始数据库。
- ConnectionString——一系列由分号（;）分开的参量，指定与数据库源连接的连接必须具备的信息。当通过 DSN 连接时，Open 方法将 DSN 的数据与脚本提供的用户名和密码一起复制到 ConnectionString 属性中。
- ConnectionTimeout——指明为了打开连接要等待多长时间（以秒计）。
- Provider——指明连接的数据提供者。
- CursorLocation——指定 ADO 在何处建立游标以便临时保存查询结果。
- IsolationLevel——指明是否可在另一事务处理内看到事务处理上的未提交的数据。
- Mode——指明为了通过连接访问数据可用的权限。模式可为未知的（在设置前是默认的）或者连接可为只读、读/写或只写访问权限。
- State——指明连接是打开的、关闭的还是正忙于连接。
- Version——返回 ADO 实现的版本号。

读者将在 463~技巧 469 内学习如何使用 ADO Command 对象的 Execute 方法发送查询以及如何使用 Connection 对象的 Execute 方法向 DBMS 发送其他（非查询的）语句。除此之外，这些技巧还向读者展示 ADO Recordset 对象如何在 SQL 的游标中处理查询返回给 ASP 脚本的多行结果集。

现在要理解的重要事情是，odbc_connect()函数和 ADO Connect 对象允许服务器端脚本使用 DSN 来建立脚本与 DBMS 之间的连接。然后脚本通过打开的连接向服务器发送命令并使用脚本来从 DBMS 提取查询结果。

技巧 460 下载、安装并使用 MyODBC 驱动程序与 MySQL 数据库连接

正如处理 ASP 和 PHP 一样，读者将发现，这些服务器端的脚本处理器的最强大的功能之

一（也是常用的）是其处理 SQL 数据库内数据的能力。可以使用嵌入 ASP 和 PHP Web 页面的脚本不仅提取并显示从 SQL 数据库提取的信息，而且还可插入、删除以及更新存储在数据库表内的信息。事实上，安装有适当的 ODBC 驱动程序之后，就可在登录到服务器或是与服务器局域网（LAN）连接的工作站之后，向 DBMS 发送任何可以执行的命令。

当上网访问自己的银行、经纪人或其他账户信息的 Web 站点、网上购物或检查定单状态时，很可能要和 SQL 数据库打交道。虽然通常在线时不会键入 SQL 的 SELECT、INSERT、UPDATE 或 DELETE 语句，但在后台，服务器端的脚本从数据库表中提取数据生成所看到的 Web 页面内容。除此之外，脚本根据访问者键入的并通过 HTML 表单向 Web 服务器发送的指令更新数据库内的数据。

大型的常设公司有足够的资金从 Microsoft、Oracle 和 IBM 这样的卖主购买最新的 SQL DBMS 产品。如果读者是一位开始创业的人或者刚刚启动了一个非商业的 Web 站点，必须在 SQL 数据库内保存由访问者提交的信息或数据，那么可考虑使用 MySQL DBMS。MySQL 是全功能的、多用户的 DBMS，可运行于许多目前流行的操作系统，如 AIX、BSDI、DEC UNIX、HP UNIX、SCO UNIXWare、Tru64 UNIX、FreeBSD、NetBSD、OpenBSD、Linux、MacOS X Server、OS/2 Warp、Solaris、Windows 95/98/ME/NT/2000/XP 上。

可从 Internet 上的 www.MySQL.com 上下载 MySQL 并免费使用。MySQL 支持所有标准的 SQL-92 数据类型、语句和事务处理。不过 MySQL 并不支持存储过程、嵌入 PHP 或 ASP Web 页面内的脚本。可以使用 MySQL 的 ODBC 接口来向 DBMS 提交一条或 SQL 语句批处理并从中提取数据。因而，可把通常放入存储过程内的语句批处理编码为脚本函数或子程序中提交给 DBMS 的语句（通过 ODBC 接口）。

Web 页面脚本必须建立与 DBMS 的连接之后才能提交查询和其他供处理的 SQL 语句。为了与 DBMS 连接，脚本必须有 ODBC 驱动程序作为中介。为了登录并打开与 DBMS 的连接，脚本把带有连接和登录细节的字符串传递给 ODBC 驱动程序。ODBC 驱动程序接着将此字符串加以格式化，使得 DBMS 可以理解，然后将此连接字符串发送给 DBMS 供处理。接下来，ODBC 驱动程序格式化由 DBMS 产生的输出并将登录企图的结果传回给脚本。

在本书写作时，为了与 MySQL DBMS 通讯脚本需要的 MyODBC 驱动程序并未与 DBMS 捆绑在一起。但是，与 MySQL DBMS 本身一样，也可从 MySQL 的 Web 站点 www.MySQL.com/Downloads/ 处下载 MyODBC（ODBC）驱动程序。只要在 downloads 的 Web 页面的 APIs（Application Program Interface）部分单击 MyODBC 超级链接。接着按照服务于不同操作系统的下载指示进行。

例如，如果在 Windows 系统上安装了 MySQL，可提取 MyODBC.zip 文件并将其保存在与安装 MySQL DBMS 同一计算机上的一个文件夹（如 C:\My Download Files）中。接着，将归档（.zip）文件中的文件提取到一个文件夹（如 C:\My Download Files\MyODBC）中，然后执行以下步骤完成安装过程：

1. 在从 MyODBC 归档文件提取的文件中找出 Setup.exe。
2. 双击 Setup.exe 启动安装程序。Setup.exe 将显示 Microsoft ODBC Setup 消息框。
3. 单击 Continue。安装程序将显示 Install Drivers（安装驱动程序）对话框。
4. 在 Available ODBC Drivers（可用的 ODBC 驱动程序）列表框中单击 MySQL，然后单击 OK 按钮。安装程序将显示 Data Sources（数据源）对话框。

5. 单击 Close 按钮。安装程序将显示 Setup Succeeded!（安装成功！）消息框。
6. 单击 OK 按钮退出安装程序。

在为 MySQL DBMS 安装了 ODBC 驱动程序（MyODBC）之后，可创建 DSN，通过它脚本可与 MySQL DBMS 通信。

当在 ODBC Data Source Administrator（ODBC 数据源管理器）的 Create New Data Source（创建新数据源）对话框中选择了 MySQL 之后，ODBC Data Source Administrator 将显示 TDX MYSQL Driver Default Configuration（TDX MYSQL 驱动程序默认设置）对话框，如图 25.12 所示。

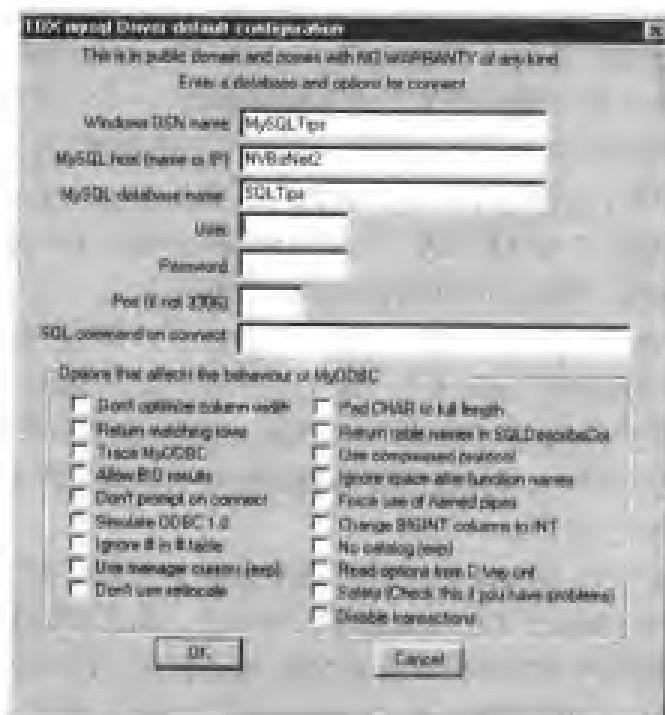


图 25.12 TDX MYSQL Driver Default Configuration 对话框

在 TDX MYSQL 对话框内只需要填充前三个域，其中包括：

- Windows DSN name——当在脚本或应用程序中引用 DSN 时想要使用的名称。由于 DSN 指向特定的数据源，如由 MySQL DBMS 管理的几个数据库之一，可键入数据库名或是数据源的单字描述。对本例来说，可在 Windows DSN Name 域内键入 MySQLTips。
- MySQL host (name or IP)——输入安装有 MySQL DBMS 的计算机的名称或 IP 地址。例如，MySQL 安装在名为 NVBizNet2 的 Windows NT 服务器上，此时可在此域内键入 NVBizNet2。
- MySQL database name——当通过已定义的 DSN 与 MySQL DBMS 连接时，脚本或应用程序想要使用的初始数据库。对本例来说，可在此域内键入 SQLTips。
- User——DSN 登录数据库时的用户名。通常应将 User 和 Password 域留为空白并让脚本在建立与 DBMS 连接时使用用户名（和密码）登录。
- Password——用户名/密码对的密码部分，是 DSN 用于登录 DBMS 时使用的。通常应将 User 和 Password 域留为空白并让脚本在建立与 DBMS 连接时使用用户名（和密

码) 登录。

- Port (if not 3306)——通常, 如果把 MySQL 安装在不允许使用端口 3306 访问的防火墙之后的计算机上, 只必须改变端口设置。在这种情况下, 必须与网络管理员联系让他打开 3306 号端口以便访问, 或是提供另一打开的端口, 然后在此域内相应地改变其值。
- SQL command on connect——每当应用程序使用正在定义的 DSN 登录到 DBMS 时, 让用户指定要执行的 SQL 语句。通常留为空白, 这个域可以用来执行一条 INSERT 语句用来记录下通过此 DSN 访问审计表的情况。

还可以使用此对话框下半部分内的复选框来设置多达 19 个不同的选项, 这些选项影响 MyODBC 驱动程序的行为。通常, 缺省设置 (没有任何一个复选框上有选择标记) 正是人们所希望的。如果在使用 MySQL 时出现了问题, 可在以后返回 ODBC 驱动程序设置屏幕, 使 Trace MyODBC 和 Safety 复选框上出现选择标记, 以便收集附加的信息, 这将有助于解决问题。

在单击 OK 按钮之后 (见图 25.12), 脚本就可使用所创建的 DSN 打开与 MySQL DBMS 的连接。例如, 在 PHP Web 页面上, 可使用技巧 459 “建立数据源 (DSN) 与 SQL DBMS 的连接”中的同一 `odbc_connect()` 函数使用在本技巧中创建的 MySQLTips DSN 与 MySQL DBMS 连接:

```
<head>
function open_DSN_connection()
(
    $conn = odbc_connect('SQLTips','Konrad','King');
    return $conn;
)
</head>
```

类似地, 如果正在使用 ASP Web 页面, 可在用 VBScript 编写的脚本内使用 ADO Connection 对象, 如下所示:

```
<%
Sub open_DSN_connection (byref connObjDSN)
    CONST dsNConnection = 'DSN=SQLTips;UID=Konrad;pwd=King;'
    'Create the ADO Connection object
    Set connObjDSN = server.createobject('adodb.connection')
    'Place the connection string into the ConnectionString
    'property within the ADO Connection object and then try
    'to establish a connection with the DBMS.
    With connObjDSN
        .ConnectionString = dsNConnection
        .open
    End With
End Sub
%>
```

技巧 461 与 MS-SQL Server 或 MySQL DBMS 建立无 DSN 的连接

通过数据源名称 (DSN) 打开嵌入 Web 页面的脚本与 SQL DBMS 之间的连接是方便的,

因为由 DSN 处理连接的细节。因而，正如读者在技巧 459 “建立数据源 (DSN) 与 SQL DBMS 的连接”中所看到的，脚本只需要提供 DSN 和合法的用户名/密码对来登录 DBMS。事实上，正如读者在技巧 460 “下载、安装并使用 MyODBC 驱动程序与 MySQL 数据库连接”的尾部所看到的示例代码一样，脚本甚至不需要了解它所连接的特定 DBMS 产品。例如，虽然在将脚本与 MS-SQL Server 连接时 ODBC 驱动程序必须传递的参数与与 MySQL 连接所需的参数是不同的，但在使用 DSN 时，脚本仍然只指定 DSN 名称和合法的用户名/密码对即可与两种 DBMS 产品中的任何一种连接。

因而通过 DSN 将脚本与 SQL 数据库连接减少了必须编写的代码量。所有的连接细节，如 DBMS 的名称和位置、所选的 ODBC 驱动程序以及会话设置等都已经编码到 DSN 中，而不是在脚本中。除此之外，使用 DSN 还让人们重用同一代码与不同的 DBMS 产品连接。

虽然方便，但使用 DSN 与 DBMS 连接也有如下几个缺点。

- 虽然脚本可提交 USE 语句选择任何由 DBMS 管理的数据库，但一旦连接起来，系统管理员必须至少为每个脚本要连接的 DBMS 创建一个 DSN。
- 当使用 DSN 而不是直接通过软件开发商提供的 OLE DB 提供程序与 DBMS 连接时，脚本要遭遇性能上的降低（OLE DB 提供程序是一种软件接口，允许外部应用程序向数据源，如 MS-SQL Server DBMS、Oracle DBMS、MySQL 数据库等，发送命令并从数据源提取数据）。由于 DSN 向 ODBC 驱动程序发送语句，后者接着将命令传递给该数据源的 OLE DB 提供程序。当向 DBMS 发送命令并从中提取数据时，使用 DSN 就意味着添加了额外的 SQL 语句和数据处理任务。

幸运的是，ADO Connect 对象允许脚本直接与某种 DBMS 的 OLE DB 提供程序连接并通信——因而避免了使用 ODBC 层时的性能降低问题。例如，为了通过无 DSN 的连接与 MySQL 连接，可使用类似以下的代码：

```
<%  
Sub open_OLEDB_connection (byref connObj)  
    connectionString = _  
        "PROVIDER=SQLOLEDB;DATA SOURCE=NVBizNet2;" & _  
        "UID=Konrad;PWD=King;DATABASE=SQLTips"  
    'Create the ADO Connection object  
    Set connObj = server.createobject("adodb.connection")  
    'Place the connection string into the ConnectionString  
    'property within the ADO Connection object and then call  
    'the .open method to establish a connection with the DBMS.  
    With connObjDSN  
        .ConnectionString = dsnConnection  
        .open  
    End With  
End Sub  
%>
```

请注意，打开无 DSN 的连接类似于打开使用 DSN 的连接——只需要改变放入连接对象的字符串即可。对于 DSN 连接，只要在连接字符串内指定 DSN、用户名和密码。作为对照，在打开无 DSN 连接时，可使用以下句法的连接字符串：


```

'PROVIDER=<name of OLE DB provider>;
DATA SOURCE=<name of the DBMS>;
UID=<username>;
PWD=<password>;
DATABASE=<initial database>'

```

虽然 MS-SQL Server 使用 OLE DB 提供程序 SQLOLEDB, 但 Oracle 使用 MSDAORA, 而 MS-Access 使用 MicrosoftJet.OLEDB.4.0。因而, 请检查一下自己的 DBMS, 看一下 OLE DB 驱动程序是什么名字。

遗憾的是, 在本书写作时, MySQL DBMS 并没有提供一个固有的 OLE DB 驱动程序, 借助于此驱动程序可使用 ADO 与 DBMS 连接。但是, 仍然可以使用 MyODBC (ODBC) 驱动程序打开与 MySQL 数据库的无 DSN 连接, 正如下面的连接字符串所表明的:

```

connectString = "DRIVER={MYSQL};SERVER=NVBizNet2;" & _
"UID=Konrad;PWD=King;DATABASE=SQLTips"

```

当然, 使用者应该用自己的在其上安装 MySQL DBMS 的服务器名代替此处的 NVBizNet2, 并对 UID 和 PWD 部分提供合法的用户名/密码对, 而且要用由 MySQL 服务器管理的一个数据库名代替此处的 SQLTips。

不管是建立有 DSN 还是无 DSN 的与 DBMS 的连接, 都将使用同一 ADO Command 和 Recordset 对象方法来处理 DBMS 内的数据。

技巧 462 使用 ADO Connection 对象执行 SELECT 语句设置访问 Web 站点的用户名/密码

当必须将 SQL 语句 (如 SELECT 语句) 发送到 DBMS 时, 可使用 ADO 的 Command 对象。例如, 假设想要设置访问 Web 站点的用户名/密码。此时可使用 HTML 表单让站点访问者键入用户名和密码。接着, 通过为 <form> 标记的 action 属性指定 ASP 或 PHP Web 页面的地址, 可让嵌入的脚本与 SQL DBMS 连接, 将键入用户名和密码的人的查询发送出去, 然后使用查询结果来决定访问者是否可以访问 Web 站点的只有成员才能访问的区域。

带有登录表单的 Web 页面可定义如下:

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<html>
<head>
  <title>Login and Start a Session</title>
</head>
<body bgcolor="lightyellow">
  <center><h1>SQL Tips and Techniques</h1>
  <hr>
  <form
    action="http://www.NVBizNet2.com/SQLTips/Login.asp"
    method="POST">
    Username: <input type="text" name="username"
               size="20"><br>
    Password: <input type="password" name="password"
               size="20"><br><br>

```

```

<input type="submit" value="Login">
<input type="reset" value="Reset">
</form>
</body>
</html>

```

在 ASP Web 页面内 (LOGIN.ASP), 可嵌入以下脚本:

```

<%
'*****
'*** OPEN DSN-Less Connection ***
'*****

Sub open_OLEDB_connection (byref connObj)
    connectString = _
        "PROVIDER=SQLOLEDB;DATA SOURCE=NVBizNet2;" & _
        "UID=sa;PWD=michele;DATABASE=SQLTips"
    With connObj
        .ConnectionString = connectString
        .open
    End With
End Sub

'*****
'*** MAIN ROUTINE ***
'*****

Dim connObj, objResultsSet, queryString
'open the connection to the DBMS
Set connObj = server.createObject ("adodb.connection")
open_OLEDB_connection (connObj)
'setup the SELECT statement to submit to the DBMS
queryString = _
    "SELECT COUNT(*) Count FROM siteAccessList " & _
    "WHERE username = '" & Request.Form("username") & _
    "' AND password = '" & Request.Form("password") & "'"
With connObj
    'submit the SELECT statement to the DBMS
    Set objResultsSet = .Execute (queryString)
    'save the session variables and then
    'move the visitor member to the member area
    If objResultsSet.Fields("count") = 1 Then
        Session("username") = Request.Form("username")
        Session("password") = Request.Form("password")
        Response.Redirect "/SQLTips/StartSession.asp"
    End If
End With

%>

```

ASP 的 REQUEST 对象的 FORM 集合允许用户将数据提取到 HTML 表单的域(称为元素)中。在本例中,脚本提取 username 和 password 元素内的输入并使两者形成如下所示的 SELECT 语句:

```
SELECT COUNT(*) FROM siteAccessList  
WHERE username='<data from username field in form>'  
AND password='<data from password field in form>'
```

在把查询字符串（也就是前面的带有来自 HTML 表单中数据的 select 语句）赋给 QUERYSTRING 变量之后，脚本使用 ADO Command 对象的 Execute 方法将 SELECT 语句（在 QUERYSTRING 变量内）发送到 DBMS 供执行。接着，DBMS 执行 SELECT 语句并将查询结果返回给 ADO Recordset 对象（在本例中是 OBJRESULTSSET）内的脚本。请注意，前面脚本中的 VBScript 语句既调用 Execute 方法又在 ADO Recordset 对象（OBJRESULTSSET）中接受查询的结果集：

```
With conObj  
    Set objResultsSet = .Execute (queryString)  
End With
```

在本例中，Recordset 对象有一个名为 COUNT 的域，其中包括由查询的 SELECT 子句中的 SQL COUNT(*) 总计函数返回的结果。登录确认脚本又来确定的 HTML 表单的输入元素内的用户名/密码对是否与保存在 SITEACCESSLIST 表的一行中的用户名和密码相匹配。如果 DBMS 返回“匹配的”行（在该种情况下 COUNT(*) 总计返回 1），所输入的用户名/密码对是合法的，脚本就把访问者重定向到 STARTSESSION.ASP Web 页面。作为对照，如果 count 域有值为 0（而不是 1），所输入的用户名/密码对是不合法的，脚本不把访问者导向到只有成员才能查看的 Web 站点。

下面的 3 个技巧将向读者展示如何通过 HTML 提交查询以及如何在 Web 页面上的 HTML 表中显示 SQL SELECT 语句返回给脚本的结果集。

技巧 463 在 Web 页面上的 HTML 表中显示查询结果

通常，人们都希望在 Web 页面上的 HTML 表内显示由 SELECT 语句返回的查询结果集。请回忆一下上次访问银行 Web 站点的情景。访问此站点很可能要查阅自己的账户余额以及储蓄的项目明细以及特定一段时间以内发出的支付支票。类似地，当访问信用卡的 Web 站点时，可以获得账户上在一段特定的支付循环中的已付及未付的清单。通过访问在线售货的 Web 站点，可以获得最近定单上的项目明细，定单何时发出，如果通过 UPS（美国特快专递服务）或是 Federal Express（联邦特快）发出的，可看到发货号。使用发货号，就可得到货物到达不同地点的日期与时间清单。在这所有情况下，由查询返回的结果集很可能显示在所查看页面的 HTML 表内。

不用编写不同的子程序来显示来自每个所提交的查询的结果集，用户可编写一个可重复使用的函数或是子程序在 HTML 表内显示查询结果集。正如下面所看到的，编写显示查询结果的脚本，而不必事先了解查询结果内的列名或列数。

ADO Recordset 对象的 Fields 集合有一个 Count 属性，可用来决定结果集内域（也就是列）的数目。为了提取结果集内返回的域（列）的数目，可以使用以下语句：

```
columnCount = objResultsSet.Fields.Count
```

接着，每个域有一个 Value 和 Name 属性，可用来提取 Fields 集合中域的名称和值（分别地）。这样一来，为了提取域的名称，可以使用以下语句：

```
columnName = objResultsSet.Fields(0).Name
```

当处理 Fields 集合内的项目时，请记住，第一个项目的索引号为 0 而不是 1。因而，前例返回 Fields 集合内第一列（索引为 0）的名称。

同时，为了提取域的值，可使用以下语句：

```
columnName = objResultsSet.Fields(0)
```

请注意，当想要提取保存在 ADO 集合内的项目中的值时，不必明显地引用 Value 属性。如果在引用集合内的项目时，忽略了属性名，缺省属性就是 Value 属性。一般来说，当引用对象及集合项目时，使用的句号 (.) 越少，则脚本的性能越好。因而，在以下两条语句中，第二条语句比第一条执行得更快，因为第二条使用了集合项目的默认属性，而不是明显地引用属性：

```
columnValue = objResultsSet.Fields(0)
```

```
columnValue = objResultsSet.Fields(0).Value
```

将这些都放在一起，就可以使用下面的 VBScript 子程序将来自于查询的结果集显示为 Web 页面上的 HTML 表：

```
Sub display_In_Table (objResultsSet)
    With Response
        .Write "<center><table border='1' cellpadding='5'>"
        .Write "<tr>"
        For column = 0 To objResultsSet.Fields.Count - 1
            .Write "<th>" & objResultsSet.Fields(column).Name _
                & "</th>"
        Next
        .Write "<tr>"
        Do While Not objResultsSet.EOF
            .Write "<tr>"
            For column = 0 To objResultsSet.Fields.Count - 1
                If objResultsSet.Fields(column) <> "" Then
                    .Write "<td>" & objResultsSet.Fields(column) _
                        & "</td>"
                Else
                    .Write "<td>&nbsp;</td>"
                End If
            Next
            .Write "<tr>"
            objResultsSet.MoveNext
        Loop
        .write "</table>"
    End With
End Sub
```

除了前面讨论过的 Fields 集合属性之外，在本例中的子程序使用了 Recordset 对象的 EOF 属性及其 MoveNext 方法在查询结果集的行内移动。MoveNext 方法将行指针移动到 ADO Recordset 对象内的下一行。当行指针超出 Recordset 对象内的最后一行时，Recordset 对象的 EOF 属性（在本例中是 OBJRESULTSSET.EOF）被设置为 TRUE（如果在 Recordset 对象内没有行，那么 SELECT 语句没有返回满足 WHERE 子句中的搜索条件的行，行指针一开始就超出 Recordset 对象内的“最后”一行，因而 Recordset 对象的 EOF 属性立即为 TRUE）。

技巧 464 编写可重用的 PHP 例程在 Web 页面上显示查询结果

在技巧 463 “在 Web 页面上的 HTML 表中显示查询结果”中，读者学习了如何使用 ADO Command 对象向 DBMS 发送查询并在 HTML 表内显示其结果集。除了 ADO 对象，某些服务器端的脚本引擎提供可用来处理不同的 DBMS 产品内的数据的函数。例如，PHP 提供用来访问保存在 dBase、Informix、InterBase、MS-SQL Server、mSQL、MySQL、Oracle、PostgreSQL、Sybase 等数据库数据的函数（除了只处理特定软件开发商的 DBMS 产品的函数之外，PHP 还提供通用的 ODBC 函数集，可用来处理任何开发商的 DBMS 产品内数据的函数）。

当使用 PHP（或其他服务器端脚本引擎）来生成 Web 页面时，可创建一套做以下事项的模块：

- 为 Web 页面写入开始和结束文本和标记块——如下例中的 STARTHTML.PHP 和 ENDHTML.PHP。
- 与 DBMS 连接——如下例中的 MSSQLCONNECT.PHP。
- 向 DBMS 发送查询——如下例中的 MSSQLQUERY.PHP。
- 在 HTML 表内显示查询结果——如下例中的 SHOWTABLE.PHP。

下面的代码展示了如何使用调用可重用模块的脚本定义 PHP Web 页面，以便在 Web 页面上显示 SQL 的查询结果：

```
<?php
include ('incFiles/StartHtml.php');
include ('incFiles/EndHTML.php');
include ('incFiles/MSSQLConnect.php');
include ('incFiles/MSSQLQuery.php');
include ('incFiles/ShowTable.php');
/** Database constants **
$db_host = 'NVBizNet2';
$db_user = 'Konrad';
$db_pass = 'King';
$db_name = 'Pubs';
/** global variables **
$link = null; //handle/channel opened to MS-SQL Server
$result = null;
/** WRITE THE TEXT AND TAGS THAT START A WEB PAGE **
startHTML ('Display Query Results',
    'Authors Table Data from the Pubs Database');
/** CALL THE ROUTINE TO CONNECT WITH THE DBMS **
if (connectToDB($db_host, $db_user, $db_pass, $db_name))
{
/** FORMULATE THE QUERY THEN PASS IT TO THE ROUTINE **
/** THAT WILL SEND IT TO THE DBMS AND DISPLAY THE **
/** QUERY RESULTS WITHIN AN HTML TABLE
$query = 'SELECT * FROM Authors '
        'ORDERED BY au_fname, au_lname';
showTable($query);
}
```

```
/** WRITE THE TEXT AND TAGS THAT END A WEB PAGE **
endHTML();
?>
```

每个 PHP 的 INCLUDE 指令（在本例中前 6 行中的每一行中）告诉 PHP 脚本引擎插入来自外部文件的内容。当想要在几个 Web 页面上重用相同的代码时，将脚本模块放入外部文件是方便的。另外，通过把代码留在外部文件而不是剪切并粘贴到其他 Web 页面中，就可通过改变单个文件（即在其他 Web 页面内使用 INCLUDE 包括进来的外部文件）一次改变几个 Web 页面的内容。

例如，假设在文件 STARTHTML.PHP 中有下面的代码：

```
<?PHP
function startHTML($title, $heading = "")
{
echo '<html><head>';
echo " <title>$title</title>";
echo '</head>
    <body bgcolor="LightYellow">
    <h1><center>SQL Tips & Techniques</center></h1>
    <hr>';
if ($heading <> "")
    echo "<h2><center>$heading</center></h2>";
return;
}
?>
```

在此文件中，可包括想让脚本在站点的每个 Web 页面的开始处放置的内容。虽然在本例中，STARTHTML()函数只插入标题和标题文本，也可容易地编写回送一条 HTML 图像标记（），从而在每个 Web 页面的顶部插入完整的标志图。

类似地，为了写入用于每个 Web 页面尾部文本内容和 HTML 标记，可使用以下保存在外部文件 ENDHTML.PHP 内的 ENDHTML()子程序的代码：

```
<?PHP
function endHTML()
{
echo '<hr>
    Created by <a href="mailto:kki@NVBizNet.com">
    Konrad King</a>.<br>
    &copy; 2002 - all rights reserved!';
echo "</body></html>";
return;
}
?>
```

在本例中，每个调用 ENDHTML()的 Web 页面将以显示 Web 管理员姓名及版权注意事项结束。为了使 Web 站点对用户更为友好，还可在每个页面底部包括指向所有站点页面的超级链接的站点地图或菜单。通过回送文件（如 ENDHTML.PHP）内的文本内容和超级链接，只需为出现在所有站点页面底部的元素键入文本、超级链接和 HTML 标记一次。类似地，将这些内容编

码到文件（如 ENDHTML.PHP）内的函数内，然后调用在 Web 页面底部写入元素的函数。

文件 MSSQLCONNECT.PHP 内的 CONNECTTODB 函数接受 MS-SQL Server (\$DB_HOST) 的名称并使用此名称和用户名 (\$DB_USER) 及密码 (\$DB_PASS) 向 DBMS 登录。在成功地登录之后，脚本将初始数据库设置为在 \$DB_NAME 中指定的数据库：

```
<?PHP
function connectToDB($db_host, $db_user, $db_pass,
                    $db_name)
{
    global $link;
    $success = true;
//If not already connected, connect to the MS-SQL Server
    if (!$link = mssql_connect($db_host, $db_user,
                              $db_pass))
    {
        $success = false;
        echo "<font color='red'><br><br><hr>".
            "<center><b>** Error ** Unable to connect ".
            "to DBMS: $db_host!</b></center>".
            "<hr><br>\n</font>";
    }
    else
    {
        //Select the database with the data you want to query
        if (!mssql_select_db($db_name, $link))
        {
            $success = false;
            echo "<font color='red'><br><br><hr>".
                "<center><b>** Error ** Unable to select ".
                "the database: $db_name!</b></center>".
                "<hr><br>\n</font>";
        }
    }
    return $success;
}
?>
```

请注意，函数 CONNECTTODB 将连接句柄（通过此句柄脚本可向 DBMS 语句并从中接受结果）放入全局变量 \$LINK 中。如果不能与 DBMS 连接，或者不能选择指定的初始数据库，该函数在 Web 页面上显示一条错误消息并返回 FALSE。主页脚本内的下列代码（以插入来自外部文件的代码的 INCLUDE 语句开始）调用子程序来查询 DBMS 并仅当 CONNECTTODB 函数能够与 DBMS 连接时才显示查询结果：

```
/** CALL THE ROUTINE TO CONNECT WITH THE DBMS **
    if (connectToDB($db_host, $db_user, $db_pass, $db_name))
    {
/** FORMULATE THE QUERY THEN PASS IT TO THE ROUTINE **
/** THAT WILL SEND IT TO THE DBMS AND DISPLAY THE **
```

```

/** QUERY RESULTS WITHIN AN HTML TABLE
    $query = "SELECT * FROM Authors ".
        "ORDERED BY au_fname, au_lname";
    showTable($query);
}
/** WRITE THE TEXT AND TAGS THAT END A WEB PAGE **
endHTML();
?>

```

如果不能连接，脚本就在每个 Web 页面底部写入内容和 HTML 标记。这样做不必调用向 DBMS 提交查询并在页面上显示查询结果集的 SHOWTABLE()函数。结果，页面既可显示查询结果的 HTML 表（如果脚本成功地与 DBMS 连接并选择包括所希望数据的数据库），也可显示错误信息（如果 DBMS 连接或数据库选择失败的话）。

外部文件 SHOWQUERY.PHP 包括调用 EXECUTEQUERY 函数（向 DBMS 提交 \$QUERY 变量中的 SELECT 语句）及在 HTML 表内显示查询结果集的脚本语句：

```

<?
function showTable($query)
{
    global $link, $result;
    //Submit SQL Query to the MS-SQL Server
    executeQuery($query);
    //determine the number of fields within in the results set
    $fields = mssql_num_fields($result);
    //display the query results set in an HTML table
    echo "<center><table border='1' cellpadding='5'>";
    //display column names as table headings
    echo "<tr>";
        for ($i=0; $i < $fields; $i++)
        {
            echo "<th>" .
                mssql_field_name($result, $i) . "</th>";
            $fieldType[$i] = mssql_field_type($result, $i);
        }
    echo "</tr>";
    //display query results (that is, the column) values
    //within the table's rows below the headings (column names)
    //that run across the top of the HTML table
    while ($array = mssql_fetch_array($result))
    {
        echo "<tr>";
        for ($i=0; $i < $fields; $i++)
        {
            if (($fieldType[$i] <> "char") and
                ($fieldType[$i] <> "blob"))
                echo "<td align='right'>";
            else

```



```

        echo '<td align="left">';
        if ($array[$i] <> null)
            echo "$array[$i]</td>";
        else
            echo "&nbsp;&nbsp;&nbsp;</td>";
        }
        echo "</tr>";
    }
    echo "</table></center>";
return;
}
?>

```

请注意，脚本检查每个域的数据类型并使非字符数据对齐于表单元格的右边。脚本将字符数据对齐于单元格的左边。

当然，仅当 EXECUTEQUERY()函数成功地向 DBMS 提交其供执行的查询时，脚本才会在 HTML 表内显示的结果集数据。正如下面的来自外部文件 MSSQLQUERY.PHP 中的代码所表明的，EXECUTEQUERY()函数既可返回查询结果集内的行，当 DBMS 由于某种原因不能执行查询时，也可显示错误消息：

```

<?
function executeQuery($query)
{
    global $link;
    global $result;
    //Submit SQL Query to the MySQL DBMS
    if (!$result = mssql_query($query, $link))
    {
        echo "<font color='red'><br><br><hr>".
            "<center><b>** Error ** The DBMS reported ".
            "an error in executing query!</b></center>".
            "<hr><br>\n</font>";
    }
    return;
}
?>

```

请注意，正如在整个脚本语句中的情况一样（无论是在主 Web 页面内还是外部文件中），EXECUTEQUERY 函数通过 \$LINK 连接句柄与 DBMS 通信（\$LINK 一开始是由外部文件 MSSQLCONNECT.PHP 内的 MSSQL_CONNECT()函数调用设置的）。

EXECUTEQUERY()函数向调用它的脚本返回查询结果集，把查询结果放入全局变量 \$RESULT 中。即使结果集由于没有与查询的 WHERE 子句中的搜索条件匹配而没有包括行，\$RESULT 仍然包括列名。因而，仅当执行查询时 DBMS 遇到了某种类型的错误，\$RESULT 才为 NULL（而 !\$RESULT 为 TRUE）。

如果正在使用的 DBMS 不是 MS-SQL Server，只要将文件内的“MSSQL_”引用改变为所使用的 DBMS 产品开头的函数调用字符串即可。例如，如果正在使用 MySQL DBMS，函数

调用将以 MySQL_（而不是 MSSQL_）开头。类似地，Oracle 函数以 Ora_ 开始，Sybase 函数以 Sybase_ 开始，依此类推。PHP 手册可从 <http://www.PHP.net/manual/en/> 处得到，其中列出并完整地描述了访问各种不同 DBMS 产品的 PHP 支持的每个函数调用。

技巧 465 通过 HTML 表单提交 SQL 查询

使用 HTML 表单提交 SQL 查询只要求使用在本书中各个技巧中所学习的有关 SELECT 语句的内容以及有关使用脚本与 SQL DBMS 连接与通讯的内容。正如读者在技巧 462“使用 ADO Connection 对象执行 SELECT 语句设置访问 Web 站点的用户名/密码”中所学习的，可以使用 HTML 表单接受来自站点访问者处的输入。在访问者单击了 HTML 表单的提交按钮之后，Web 浏览器将表单元素内访问者的输入和选择发送给 HTML 表单的 <form> 标记的 action 属性给定的 Web 地址（也就是 URL）。

当表单的 action 属性指明 PHP 或 ASP Web 页面的 URL 时，嵌入页面的脚本可通过名称提取访问者在 HTML 表单上的输入。例如，假设创建了一个如图 25.13 所示的“查询”页面，而表单定义如下：

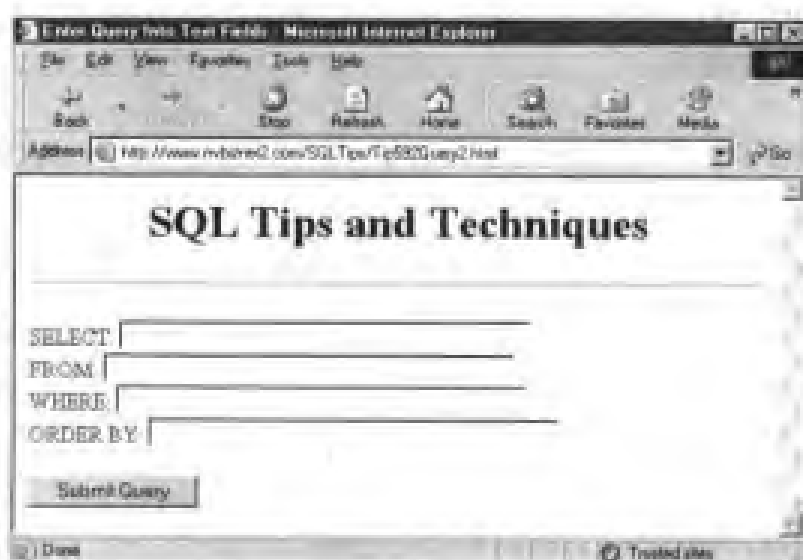


图 25.13 带有访问者可提交 SQL 查询的 HTML 表单的 Web 页面

```
<form action='http://www.NVBizNet2.com/SQLTips/Query.php'
      method='POST'>
  SELECT: <input type='text' name='selectClause'
           size='40'><br>
  FROM: <input type='text' name='fromClause' size='40'><br>
  WHERE: <input type='text' name='whereClause'
         size='40'><br>
  ORDER BY: <input type='text' name='orderBy'
            size='40'><br><br>
  <input type='submit' value='Submit Query'>
</form>
```

在本例中，PHP 处理器将访问者输入到表单元素中的数据复制到 PHP Web 页面内的脚本 QUERY.PHP 可访问的变量中。每个带有来自表单元素数据的变量名称由元素名前面加美元符

(`$`) 所组成。因而, `QUERY.PHP` 内的脚本可从变量 `$selectClause` 中提取输入到表单的 `selectClause` 元素中的文本, 而从变量 `$fromClause` 中提取输入到表单的 `fromClause` 元素中的文本, 从变量 `$whereClause` 中提取输入到表单的 `whereClause` 元素中的文本, 从变量 `$orderBy` 中提取输入到表单的 `orderBy` 元素中的文本。

为了向 DBMS 提交访问者在 HTML 表单的 4 个域内输入的内容定义的 SQL 查询, 可定义 PHP Web 页面如下:

```
<?PHP
    include ('incFiles/MSSQLConnect.php');
    include ('incFiles/MSSQLQuery.php');
    include ('incFiles/StartHtml.php');
    include ('incFiles/EndHTML.php');
    include ('incFiles/ShowTable.php');
/** database connection constants **
    $db_host = "NVBizNet2";
    $db_user = "konrad";
    $db_pass = "king";
    $db_name = "pubs";
/** global variables **
    $link = null; //handle/channel opened to the DBMS
    $result = null;
/*******
/** Put the form data into an HTML table display **
/** as a heading before the table of query results **
/*******
    $headingQuery =
        "<table border='0' >" .
        "<tr><td><b>SELECT</b> " . $selectClause . "</td></tr>"
        "<tr><td><b>FROM</b> " . $fromClause . "</td></tr>";
    IF (trim($whereClause) <> "")
        $headingQuery .=
            "<tr><td><b>WHERE</b> " . $whereClause . "</td></tr>";
    If (trim($orderBy) <> "")
        $headingQuery .=
            "<tr><td><b>ORDER BY</b> " . $orderBy . "</td></tr>";
    $headingQuery = str_replace("\\", "", $headingQuery);
/*******
/** Display the Web Page Title **
/*******
    startHTML ("Display Query Results", $headingQuery);
/*******
/** Connect to the DBMS, and call showTable, which **
/** submits the query and displays the query results **
/*******
    if (connectToDB($db_host, $db_user, $db_pass, $db_name))
    {
```

```

$query = "SELECT " . $selectClause . " " .
        "FROM " . $fromClause;
IF (trim($whereClause) <> "")
    $query .= " WHERE " . $whereClause;
If (trim($orderBy) <> "")
    $query .= " ORDER BY " . $orderBy;
$query = str_replace("\\" , "", $query);
showTable($query);
}
endHTML();
?>

```

类似地，如果表单的 action 属性指明了 ASP Web 页面的 URL，如 <http://www.NVBizNet2.com/SQLTips/Query.ASP>，可使用下面的 VBScript 形成并提交基于访问者输入到 HTML 表单中的内容的查询：

```

<%
Dim connObj, objResultsSet, queryString
'open the connection to the DBMS
Set connObj = server.createObject ("adodb.connection")
open_OLEDB_connection (connObj)
queryString = "SELECT " & Request.Form("selectClause") & _
"FROM " & Request.Form("fromClause")
If Trim(Request.Form("whereClause")) <> "" Then
queryString = _
queryString & " WHERE " & Request.Form("whereClause")
End If
If Trim(Request.Form("orderBy")) <> "" Then
queryString = _
queryString & " ORDER BY " & Request.Form("orderBy")
End If
'submit the SELECT statement (the query) to the DBMS
Set objResultsSet = connObj.Execute (queryString)
'call the routine that displays the query results set
'within an HTML table on the Web page
display_In_Table (objResultsSet)
%>

```

代码被使用但未在此处重复的是子程序 open_OLEDB_connection()（在技巧 461 “与 MS-SQL Server 或 MySQL DBMS 建立无 DSN 的连接”中显示的）以及 display_In_Table()（在技巧 463 “在 Web 页面上的 HTML 表中显示查询结果”中显示的）内的代码。

请注意，VBScript 使用 Request 对象的 Form 集合允许提取输入到 HTML 表元素中的内容。Form 集合是一个数组（在 Request 对象内），ASP 脚本宿主将访问者在 HTML 表单上的输入和选择复制其内，而 Web 浏览器要把其中的数据发送到 ASP Web 页面供处理。正如本例所表明的，脚本通过引用 Form 集合内的相同名称可提取输入到每个表元素中的值。

在让站点访问者向 DBMS 提交 SQL 查询时，前面的图 25.13 中所显示的 Web 页面有想要显示的最少内容。但是，只是显示访问者可输入 SELECT 语句的各子句的表单并不是对所有

用户都是友好的。为了使用查询表单，访问者不仅必须了解数据库内的表名，而且还有表列的名称。除此之外，访问者必须了解 SQL SELECT 语句内每个子句的正确句法。

为用户所创建的查询表单（通常管理员通过公司的内部网访问 DBMS）应该看起来更像图 25.14 那样才好。

SQL Tips and Techniques		
<input type="checkbox"/> Title ID	<input type="checkbox"/> TitleAuthor	<input type="checkbox"/> Authors
<input type="checkbox"/> Title	<input type="checkbox"/> Title ID	<input type="checkbox"/> First Name
<input type="checkbox"/> Type	<input type="checkbox"/> Royalty Percent	<input type="checkbox"/> Last Name
<input type="checkbox"/> Publisher ID		<input type="checkbox"/> Phone
<input type="checkbox"/> Price		<input type="checkbox"/> Address
<input type="checkbox"/> Advance		<input type="checkbox"/> City
<input type="checkbox"/> Royalty		<input type="checkbox"/> State
<input type="checkbox"/> PTDOrder		<input type="checkbox"/> Zip Code
<input type="checkbox"/> Status		
<input type="checkbox"/> Date Published		
ORDER BY _____		
<input type="button" value="Submit Query"/>		

图 25.14 访问者可在其上输入并选择数据以形成并提交 SQL 查询的 Web 页面

如图 25.14 所示的 Web 页面是对用户更为友好的。访问者不再必须了解数据库内表的名称和结构才能提交查询。而是访问者只要单击 DBMS 要报告值的各列对应的复选框即可。根据所选的列，形成查询的脚本，即把表名插入 SELECT 语句的 FROM 子句，把列名放入查询的 SELECT 子句。表单底部的单选按钮帮助访问者指定在选择添加到结果集中的行时 DBMS 使用的搜索条件。

创建带 HTML 表单的 Web 页面，允许用户向 DBMS 提交查询常常涉及创建用户友好的 Web 界面和允许用户编写涉及多表、列和搜索条件的查询之间的折衷。例如，图 25.13 中的 HTML 表单，虽然不是怎么用户友好，但却允许用户提交涉及任何数目的表、列和搜索条件的查询。作为对照，图 25.14 中的 Web 页面中的 HTML 表单需要的有关数据库结构以及编写 SQL SELECT 语句的知识较少，但是，在本例中的表单却把用户的选择局限于对 3 个表中的列，而且在 SELECT 语句的 WHERE 子句中至多有 4 个搜索条件。

当然，所创建的 HTML 表单依赖于用户对系统的知识和查询需求。当看到自己必须为资深用户使用编写一套 Web 页面的界面页面，其中必须查看数据库内来自不同方面的数据，而又要为经理们编写另一套 Web 页面，他们需要按每日、每周或每月为基础的特定的预先定义的报告时，也没有什么可大惊小怪的。

要理解的重要事情是，可以使用 HTML 表单让 DBMS 用户（或 Web 站点访问者）指定想要搜索 DBMS 获得的数据。使用服务器端的脚本语言，如 VBScript、PHP、JScript 等，可以根据用户在 HTML 表单中的输入编写 SELECT 语句，将查询提交给 DBMS，然后为用户在 Web 页面上的 HTML 表内显示查询结果集。

技巧 466 使用 HTML 表单向 SQL 表中插入数据

除了允许访问者使用表单编写 SQL 查询之外，还可使用表单让访问者插入、更新或是删除数据库中的数据。事实上，通过 HTML 表单处理数据库更新涉及访问者与脚本以及脚本与 DBMS 之间的同样的通信问题。

在 PHP 脚本之内，通过从与表单元素的同名变量中提取表单数据来处理基于 HTML 表单的数据库更新。类似地，在 VBScript 中，是从 Request 对象的 Form 集合中访问表单元素，脚本向 DBMS 提交所希望执行的 INSERT、UPDATE 或 DELETE 语句。

例如，假想想让销售人员在线维护有关其顾客的信息。此时可以使用类似于图 25.15 所示的表单向销售人员的顾客清单中添加新顾客。

The image shows a screenshot of a web browser window titled "Add Customer Data - Microsoft Internet Explorer". The address bar shows the URL "http://www.nvbiznet2.com/SQLTips/AddCust.asp". The main content area displays a form titled "Add New Customer". The form contains the following elements: a "Salesrep" dropdown menu with "Kornel King" selected; "Customer" section with "First", "Last", "Street", "City", "State", and "Zip Code" text input fields; and a "Phone" text input field. At the bottom of the form are two buttons: "Insert" and "Clear". The browser's status bar at the bottom indicates "Done" and "100% Zoom".

图 25.15 插入并（或）更新顾客数据的 HTML 表单

为了提取输入到 HTML 表单中的顾客信息并将其插入到 CUSTOMERS 表中，可将 HTML 表单的 <form> 标记的 action 属性设置为 ASP Web 页面的 URL，如下所示：

```
<form  
  action='http://www.NVBizNet2.com/SQLTips/AddCust.asp'  
  method='POST'>
```

接着，在 ASP Web 页面内（在本例中是 ADDCUST.ASP）嵌入如下的 VBScript 代码：

```
<%  
Dim connObj, objResultSet, statementString  
'create a connection object and then call a subroutine  
'to open a connection to the DBMS  
Set connObj = server.createObject ("adodb.connection")  
open_OLEDB_connection (connObj)  
'Formulate the INSERT statement based on the visitor's  
'inputs within the HTML form  
statementString = _  
  'INSERT INTO customers ' & _
```

```

" (first_name, last_name, street_addr, city, state, " & _
" zip_code, phone_number, salesrep_ID) VALUES (" & _
" '" & Request.Form("fName") & "' & _
" ', '" & Request.Form("lName") & "' & _
" ', '" & Request.Form("stAddress") & "' & _
" ', '" & Request.Form("city") & "' & _
" ', '" & Request.Form("state") & "' & _
" ', '" & Request.Form("zipCode") & "' & _
" ', '" & Request.Form("phoneNumber") & "' & _
" ', '" & Request.Form("salesrepID") & "' & _
'submit the INSERT statement to the DBMS for execution
connObj.Execute statementString,,adExecuteNoRecords
%>

```

在本例中的 VBScript 调用 `open_OLEDB_connection()` 子程序(读者在技巧 461“与 MS-SQL Server 或 MySQL DBMS 建立无 DSN 的连接”中学习过有关内容) 打开与 DBMS 的连接。为了构建 SQL INSERT 语句,脚本使用 Request 对象的 Form 集合来提取输入到表单中的数据(如图 25.15 所示)。最后,为了向 DBMS 提交供执行的 INSERT 语句,此 VBScript 使用 ADO Connection 对象的 Execute 方法。请注意,当使用 Execute 方法提交 INSERT、DELETE 或 UPDATE 语句时,不必期望 DBMS 返回结果集。因此,通过将 Execute 方法调用的句法改变为以下形式,可避免创建 Recordset 对象的开销:

```

<connection object>.EXECUTE <SQL statement>,ra,options
其中:

```

- <SQL statement>是 DBMS 执行的 SQL INSERT、DELETE 或 UPDATE 语句。
- ra 是可选的参数,可指明查询影响的行数。
- options 指明 DBMS 如何执行<SQL statement>参数内传递来的 SQL 语句。

在本例中的 Execute 方法调用告诉 DBMS 执行 SQL 语句 (STATEMENTSTRING 内的), 而不返回任何结果集中的记录。

```

connObj.Execute statementString,,adExecuteNoRecords

```

注意: 对于枚举值的清单(如 `adExecuteNoRecords`), 可用来设置各种 ADO 对象及方法的选项和属性(如 Connect 对象的 Execute 方法, 请访问 <http://www.w3schools.com/ado/> 站点并单击列在 Web 页面左边的 ADO 对象之一。在 Web 浏览器显示有关该 ADO 对象的 W3Schools 信息之后, 可单击对象属性或方法之一的超级链接。如果该方法或属性有枚举的值(也就是命名常数), 可用来设置其选项或属性, 此时可发现列在下面的说明如何使用属性或方法的示例代码中的枚举值。

为了得到所有 ADO 对象属性和方法的完整的所有枚举值的清单, 可访问 Microsoft Developer Network (MSDN) 数据库, 其网址为 <http://msdn.microsoft.com/library/>。为了找到 ADO 枚举类型清单, 在屏幕左边的菜单中, 作出如下选择:

1. 单击 Data Access (数据访问) 左边的加号 (+)。
2. 在 Data Access 菜单中, 单击 Microsoft Data Access Components (MDAC) 左边的加号 (+)。
3. 在 MDAC 菜单中, 单击 SDK Documentation (软件开发工具箱文档) 左边的加号 (+)。

4. 在 SDK Documentation 菜单中, 单击 Microsoft ActiveX Data Objects (ADO) 左边的加号 (+)。

5. 在 ADO 菜单中, 单击 ADO Programmer's Reference (ADO 编程人员参考) 左边的加号 (+)。

6. 在 ADO Programmer's Reference 菜单中, 单击 ADO API Reference (ADO API 参考) 左边的加号 (+)。

7. 在 ADO API Reference 菜单中, 单击 ADO Enumerated Constants (ADO 枚举常数)。

注意: 单击词汇 ADO Enumerated Constants 而不是其左边的加号 (+)。

在选择了 ADO Enumerated Constants 之后, MSDN 站点将在其右边显示一带指向各个常数的超级链接的页面。由于到达此处涉及一系列选择, 一定要把此页面添加到浏览器的收藏夹中, 以便今后可单击一次即可返回该页面。

技巧 467 通过 HTML 表单更新及删除数据库数据

在更新或删除数据库内数据之前, 必须首先搜索带有想要改变数据的列的行或是想要删除的行。接着, 执行 UPDATE 或 DELETE 语句来分别改变或删除数据。当然执行搜索和更新以及搜索和删除都是在同一语句内进行。UPDATE 或 DELETE 语句内的 WHERE 子句包括 DBMS 用来识别要删除的目标行或有要改变数据的目标行的搜索条件。在 UPDATE 语句中, SET 子句指定要改变的列值。

下面的表单为更新表中的记录提供最大的灵活性:

```
<form action='http://www.NVBizNet2.com/SQLTips/Update.asp'  
  method='POST'  
  UPDATE: <input type='text' name='tableName'  
    size='40'><br>  
  SET: <input type='text' name='setClause' size='40'><br>  
  WHERE: <input type='text' name='whereClause'  
    size='40'><br>  
  <input type='submit' value='Update Record'>  
</form>
```

为了处理更新表单, 应在 ASP Web 页面 (在本例是 UPDATE.ASP) 内嵌入如下 VBScript:

```
<%  
Dim connObj, objResultSet, statementString  
'create a connection object and then call a subroutine  
'to open a connection to the DBMS  
Set connObj = server.createObject ('adodb.connection')  
open_OLEDB_connection (connObj)  
'Formulate the UPDATE statement based on the visitor's  
'inputs within the HTML form  
statementString = _  
  'UPDATE ' & Request.Form('tableName') & _  
  ' SET ' & Request.Form('setClause') & _  
  ' WHERE ' & Request.Form('whereClause')  
'submit the UPDATE statement to the DBMS for execution
```



```
connObj.Execute statementString,,adExecuteNoRecords
%>
```

类似地，可使用如下表单删除表中的行：

```
<form action="http://www.NVBizNet2.com/SQLTips/Delete.asp"
      method="POST">
  DELETE: <input type="text" name="tableName"
           size="40"><br>
  WHERE: <input type="text" name="whereClause"
          size="40"><br>
  <input type="submit" value="Delete Record">
</form>
```

为了处理删除表单，可将如下的 VBScript 嵌入 ASP Web 页面(在本例中是 DELETE.ASP)：

```
<%
Dim connObj, objResultsSet, statementString
'create a connection object and then call a subroutine
'to open a connection to the DBMS
Set connObj = server.createobject ("adodb.connection")
open_OLEDB_connection (connObj)
'Formulate the DELETE statement based on the visitor's
'inputs within the HTML form
statementString = _
  "DELETE FROM " & Request.Form("tableName") & _
  " WHERE " & Request.Form("whereClause")
'submit the DELETE statement to the DBMS for execution
connObj.Execute statementString,,adExecuteNoRecords
%>
```

虽然本技巧中前面示例中的表单使得更新或删除 DBMS 表中的行变得容易，但用户必须了解如何编写 SQL DELETE 和 UPDATE 语句，才能使用这两个表单。除此之外，如果用户向两个表单的 whereClause 元素中输入了错误的条件，他或她可能会不经意间删除或更新表中错误的、太多的或所有的行。因此，应该建立删除或更新访问，使得在用户执行显示目标行的查询之后才出现删除或更新。接着让用户在屏幕上选择要 UPDATE 或 DELETE 的行。

例如，通过 HTML 表单建立对 CUSTOMERS 表的 DELETE 访问，可使用如下的查询表单让用户显示他或她可删除的顾客清单：

```
<form
'  action="http://www.NVBizNet2.com/SQLTips/DelCustSel.asp"
  method="POST">
  SELECT: cust_ID, <input type="text" name="selectClause"
           size="40"><br>
  FROM: <input type="text" name="fromClause" size="40"
        value=customers593><br>
  WHERE: <input type="text" name="whereClause"
          size="40"><br>
  ORDER BY: <input type="text" name="orderBy"
             size="40"><br><br>
  <input type="submit" value="Submit Query">
```

</form>

请注意，在本例中的表单强制用户在查询的 SELECT 子句中他或她决定要显示的列清单中包括 CUST_ID 列。对于本例中的 CUSTOMERS 表，CUST_ID 是 PRIMARY KEY，为了删除在表单上标记为删除的顾客，查询结果必须包括有一列，从 CUSTOMERS 表删除行的 VBScript 可用该列来识别 CUSTOMERS 表那些用户标记为要删除的行。

当用户单击本例中的 Submit(提交)按钮(标记为 Submit Query)时，下面的嵌入 ASP Web 页面(DEL CUSTSEL.ASP)的子程序将与用户的搜索条件匹配的顾客清单呈现出来，如图 25.16 所示。



图 25.16 此 HTML 表单中有显示 CUSTOMERS 表查询结果集的复选框，也有用户用于标记要删除的顾客的复选框

```
Sub display_In_Table (objResultSet)
    With Response
        .Write "<form action=' & _
        '<br>'http://www.NVBizNet2.com/SQLTips/DelCustRows.asp' & _
        "<br>'method='POST'>"
        .Write "<table border='1' cellpadding='5'>"
        .Write "<tr>"
        .Write "<td>DEL</td>"
        For column = 0 To objResultSet.Fields.Count - 1
            .Write "<th>" & objResultSet.Fields(column).Name & _
            "<br>" & "</th>"
        Next
        .Write "<tr>"
        DIM row_number
        row_number = 0
        Do While Not objResultSet.EOF
            .Write "<tr>"
```

```

        row_number = row_number + 1
'place a check box within the first column of each row
        .Write "<td><input type='checkbox' name='row' & _
                row_number & "' Value='" & _
                objResultsSet.Fields("cust_ID") & "'></td>"
'Display column values from the SQL table
        For column = 0 To objResultsSet.Fields.Count - 1
            If objResultsSet.Fields(column) <> "" Then
                .Write "<td>" & objResultsSet.Fields(column) _
                    & "</td>"
            Else
                .Write "<td>&nbsp;</td>"
            End If
        Next
        .Write "<tr>"
        objResultsSet.MoveNext
    Loop
    .write "</table><br>"
    .Write "<input type='submit' value='Delete Rows'> " & _
        "<input type='reset' value='Reset'>"
    .Write "</form>"
End With
End Sub

```

请注意，本例中展示的 `display_In_Table()` 子程序在 HTML 表单内的 HTML 表内显示查询结果。表内每行的第一个单元格包括一个复选框，用户可单击此复选框来标记 CUSTOMERS 表中要删除的行。

当用户单击此表单内标记为 Delete Rows 的提交按钮时，Web 浏览器向 ASP Web 页面（DELCASTROWS.ASP）上的脚本发送用户用复选框所做出的选择。DELCASTROWS.ASP 接着使用如下脚本从 CUSTOMERS 表中删除前面图 25.16 中的表单中的复选框所选定的行：

```

<%
Dim connObj, statementString, i
'open the connection to the DBMS
Set connObj = server.createobject ("adodb.connection")
open_OLEDB_connection (connObj)
'setup each DELETE statement to submit to the DBMS
For i = 1 To Request.Form.Count
    statementString = "DELETE FROM CUSTOMERS593 " & _
        " WHERE cust_ID=" & Request.Form(i)
'submit the DELETE statement to the DBMS
    connObj.Execute statementString,,adExecuteNoRecords
Next
'return to the original CUSTOMERS search page
Response.Redirect "DeleteCust.HTML"
%>

```

技巧 468 从脚本内调用存储过程

只要可能，就应该使用存储过程来代表嵌入 Web 页面的脚本来执行 SQL 语句。使用存储过程可导致 DBMS 性能提高及更快的语句执行速度。

在执行通过 ADO Connect 或 Command 对象的 Execute 方法提交的 SQL 语句时，DBMS 必须首先生成执行计划。作为对照，当脚本调用存储过程时，DBMS 立即执行存储过程的语句。由于 DBMS 在执行 CREATE PROCEDURE 语句时创建存储过程的执行计划，则 DBMS 避免了为存储过程中的语句批处理多次生成同样的计划。不必生成执行计划就减少了 DBMS 上的处理负载。另外，DBMS 能够快速执行语句，因为它不必花时间来为存储过程语句创建实时的执行计划。

除此之外，使用存储过程还可保证插入、更新或删除过程中的所有步骤都要完成。例如，假设有一个顾客可调用的（通过 Web 页面上的超级链接）VBScript 来取消一个定单。除了从 ORDERS 表中删除定单之外，脚本还必须删除该定单的 ORDERDETAILS 行（从而不会在 ORDERDETAILS 表中留下孤行），更新 COMMISSIONS 表，从而使得销售人员不会从取消的定单上获得奖金，而且必须将货品再添加到 INVENTORY 表中。不是将这些语句分别编入一个脚本中，让脚本调用一个存储过程来执行所有所需的语句就不会不小心忘记一个或多个步骤，可能更为容易（也更可靠）（确实如此，脚本总是要执行编入其中的所有语句。但是，编程人员后来更新 Web 页面可能会意外地忽略一行代码，或者在脚本进程与 DBMS 之间通过网络或 Internet 通讯发送多个语句时出了故障）。

为了在脚本内调用存储过程，只要向 ADO Connect 对象的 Execute 方法传递想要调用的存储过程名来代替 SQL 语句。例如，假设顾客使用 Web 页面上的 HTML 表单取消一个定单。如果没有存储过程，脚本必须调用 Connection（或 Command）对象的 Execute 方法 4 次，从而更新 ORDERS、ORDERDETAILS、COMMISSIONS 和 INVENTORY 表，以反映定单的取消。通过依赖于存储过程来更新 4 个表，脚本只需调用 Execute 方法一次，如下所示：

```
<%  
Dim connObj, statementString  
'Formulate the statement that calls the stored procedure  
statementString = "usp_cancel_order " & _  
    " @order_number=" & Request.Form("order_number") & "'  
'open the connection to the DBMS and execute the stored  
'procedure call  
Set connObj = server.createObject ("adodb.connection")  
open_OLEDB_connection (connObj)  
connObj.Execute statementString,,adExecuteNoRecords  
>%
```

请注意，脚本在必要时可将参数值传递给存储过程。在本例中，脚本使用 Request 对象中的 Form 集合来提取顾客在 Web 页面的 HTML 表单中输入的（或选择的）定单号。接着脚本创建语句字符串，通过 @ORDER_NUMBE 参数将定单号传递给存储过程。

虽然前例中的存储过程没有返回查询结果，但脚本可调用存储过程执行查询并返回结果集，只要创建一个 DBMS 可在其中放入存储过程返回的查询结果的 Recordset 对象，如下所示：

```
<%  
Dim connObj, objResultsSet, statementString
```

```

'Formulate the statement that calls the stored procedure
statementString = 'sales_by_year ' & _
    ' @beginning_date=' & Request.form('start_date') & _
    ' ', @ending_date=' ' & Request.form('end_date') & ' '
'open the connection to the DBMS
Set connObj = server.createObject ('adodb.connection')
open_OLEDB_connection (connObj)
'call the stored procedure that executes the query
Set objResultsSet = connObj.Execute (statementString)
'display the query results returned by the stored procedure
'within an HTML table
display_In_Table (objResultsSet)
?>

```

技巧 469 使用 VBScript 处理 Recordset 对象

在技巧 467 “通过 HTML 表单更新及删除数据库数据”中，读者学习如何创建 HTML 表单，让用户生成并执行 SQL UPDATE 和 DELETE 语句。读者还了解到，让大多数用户删除表行或改变数据库内的数据的最好方式是要求他们首先执行一条在屏幕上显示潜在目标行的查询。接着，由于数据显示在 HTML 表单之内，用户可可视化地选择要改变或删除的行，这是比选择 SQL UPDATE 或 DELETE 中的 WHERE 子句中的搜索条件更为容易也较不易出错。虽然用户有时可能想要同时删除几个表行，但修改表内的列值通常是一次一行的操作。因而，可能要用从数据库表中的特定行中提取列值的超级链接来代替技巧 467 中每个查询结果行开始处的复选框。通过代替 HTML 表单元元素内的行列值，可让用户按要求修改行内的每个值。当完成修改行值之后，用户单击 HTML 表单的 Submit 按钮将表单结果发送到一个服务器端的脚本中，从而向 DBMS 提交供执行的 UPDATE 语句。

例如，假设想要创建一个基于 Web 的应用程序，让用户更新 CUSTOMERS 表内的数据。首先，创建一个 Web 页面，其中有如下的表单，让用户选择 CUSTOMERS 表内想要改变的行：

```

<form
action="http://www.NVBizNet2.com/SQLTips/CustList.asp"
method="POST">
  SELECT: cust_ID, <input type="text" name="selectClause"
              size="40"><br>
  FROM: <input type="text" name="fromClause" size="40"
        value=customers593><br>
  WHERE: <input type="text" name="whereClause"
        size="40"><br>
  ORDER BY: <input type="text" name="orderBy"
        size="40"><br><br>
  <input type="submit" value="Submit Query">
</form>

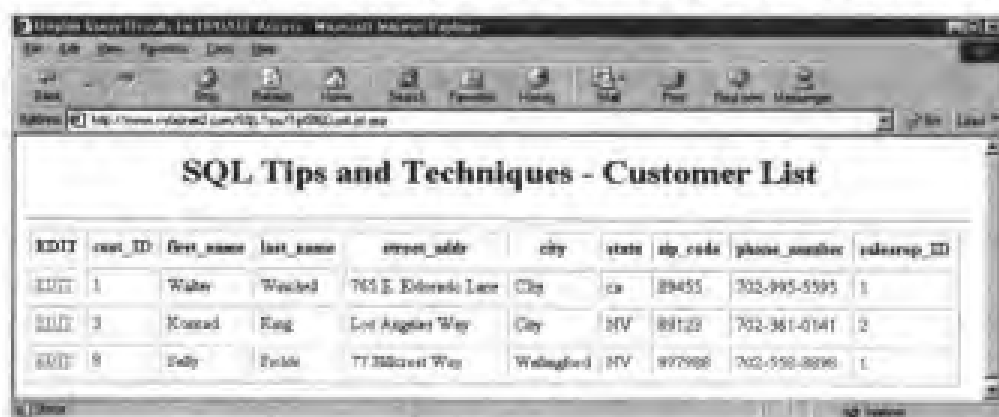
```

在用户单击了表单的标记为 Submit Query（提交查询）的提交按钮后，Web 浏览器向 ASP（或 PHP）Web 页面（在本例中是 CUSTLIST.ASP）发送表单结果（也就是输入到表单中的信息）。嵌入 CUSTLIST.ASP 的是下面的脚本，根据表单结果创建 SQL SELECT 语句并向 DBMS

提交查询:

```
<%
Sub SubmitQuery(objConn, byref objRecordset)
    DIM queryString
    If (Trim(Request.Form('selectClause')) = '') Then
        queryString = 'SELECT * '
    else
        queryString = 'SELECT cust_ID '
        If (Trim(Request.form('selectClause')) <> '') Then
            queryString = queryString & ', ' &
                Trim(Request.form('selectClause'))
        End If
    End If
    queryString = _
        queryString & ' FROM ' & Request.Form('fromClause')
    If Trim(Request.Form('whereClause')) <> '' Then
        queryString = queryString & ' WHERE ' & _
            Request.Form('whereClause')
    End If
    If Trim(Request.Form('orderBy')) <> '' Then
        queryString =
            queryString & ' ORDER BY ' & Request.Form('orderBy')
    End If
    'submit the query, the SELECT statement to the DBMS
    Set objRecordset = objConn.Execute (queryString)
End Sub
%>
```

请注意，脚本必须在查询的 SELECT 子句中包括来自目标表中的 PRIMARY KEY 列（在本例中是 CUST_ID）。在更新过程中，其他脚本将使用 PRIMARY KEY 值来提取然后修改目标表（在本例中是 CUSTOMERS）内特定行中的值。在提交查询之后，CUSTLIST.ASP Web 页面内的另一脚本必须处理由 DBMS 返回的查询结果。DisplayInTable()是 VBScript 子程序，用来处理在 ADO Recordset 对象中返回的查询结果行，以便显示如图 25.17 所示的顾客清单。



EDIT	cust_ID	first_name	last_name	street_address	city	state	zip_code	phone_number	salesrep_ID
EDIT	1	Walter	Wesley	765 E. Edwards Lane	City	ca	89455	702-995-5595	1
EDIT	3	Kenned	Rag	Los Angeles Way	City	NV	89129	702-361-0141	2
EDIT	8	Felly	Felds	77330 West Way	Wabagford	NV	897988	702-558-8898	1

图 25.17 HTML 表单，允许用户单击第一列内的 Edit 超级链接，从而选择 CUSTOMERS 表内要更新的行

```

<%
Sub DisplayInTable(objRecordset)
    With Response
        .Write "<table border='1' cellpadding='5'>"
        .Write "<tr>"
        'use Recordset field names as HTML table column headings
        .Write "<th>EDIT</th>"
        For column = 0 To objRecordset.Fields.Count - 1
            .Write "<th>" & objRecordset.Fields(column).Name _
                & "</th>"
        Next
        .Write "<tr>"
        'display the value in the Recordset within the HTML table
        Do While Not objRecordset.EOF
            .Write "<tr>"
            'Put an "EDIT" hyperlink in the First column of each row
            .Write "<td><a href='EditCust.asp?cust_ID=" & _
                objRecordset.Fields("cust_ID") & "'>EDIT</a></td>"
            For column = 0 To objRecordset.Fields.Count - 1
                If objRecordset.Fields(column) <> "" Then
                    .Write "<td>" & objRecordset.Fields(column) _
                        & "</td>"
                Else
                    .Write "<td>&nbsp;</td>"
                End If
            Next
            .Write "<tr>"
            objRecordset.MoveNext
        Loop
        .write '</table>'
    End With
End Sub
%>

```

在用户单击了图 25.17 中表的第一列内的 Edit 超级链接之后, Web 浏览器提取 ASP Web 页面 (EDITCUST.ASP) 并向其传递带有 CUSTOMERS 表内某一顾客行的 PRIMARY KEY 值的查询字符串。嵌入 EDITCUST.ASP 内的 VBScript 脚本使用 PRIMARY KEY 值来提取顾客行并调用下面的 DisplayInForm() 子程序在 HTML 表单内显示行的当前列值:

```

<%
Sub DisplayInForm(objRecordset)
    With Response
        .Write _
            "<form action=" & _
            "'http://www.NVBizNet2.com/SQLTips/UpdateCust.asp'" & _
            "method='POST'" >"
        .Write "<table border='1' cellpadding='5'>"
        .Write "<tr>"
    End With
End Sub
%>

```

```

'Use Recordset field names as HTML table column headings
For column = 0 To objRecordset.Fields.Count - 1
    .Write "<th>" & objRecordset.Fields(column).Name _
        & "</th>"
Next
.Write "<tr>"
'display the value in the Recordset within the HTML table
Do While Not objRecordset.EOF
    .Write "<tr>"
'Put an "EDIT" hyperlink into the first column of each row
For column = 0 To objRecordset.Fields.Count - 1
    .Write "<td>" & _
        "<input type='text' " & _
        "name='" & _
        objRecordset.Fields(column).Name & "' " & _
        "size='" & _
        objRecordset.Fields(column).ActualSize & "' " & _
        "value='" & objRecordset.Fields(column) & _
        "' " & "</td>"
Next
    .Write "<tr>"
    objRecordset.MoveNext
Loop
.write "</table>"
.Write "<input name='primaryKey'
type='hidden' value='" & _
Request.QueryString("cust_ID") & "'>"
.Write "<br><input type='submit' value='Save Changes'>"
.Write " <input type='reset' value='Reset'>"
.Write "</form>"
End With
End Sub
%>

```

当用户单击表单的提交按钮（标记为 Save Changes[保存变化]）时，Web 浏览器向嵌入 ASP Web 页面的 UPDATECUST.ASP 发送表单结果。嵌入 UPDATECUST 内的 SubmitUpdate() 子程序创建并向 DBMS 提交 UPDATE 语句，向 CUSTOMERS 表的顾客行写入新列值：

```

<%
Sub SubmitUpdate(objConn)
    DIM queryString, i, setCount
    setCount = 0
    queryString = "UPDATE customers593 SET "
    For i = 1 To Request.Form.Count - 1
        If Request.Form.Key(i) <> "cust_ID" Then
            setCount = setCount + 1
            If setCount > 1 Then
                queryString = queryString & ","
            End If
            queryString = queryString & Request.Form(i) & "=" & Request.Form(i) & ","
        End If
    Next
    queryString = queryString & ";"
    objConn.Execute(queryString)
End Sub
%>

```



```
End If
    queryString = _
        queryString & Request.Form(key(i)) & '=' & _
        Request.Form(i) & ''
End If
Next
queryString = queryString & "WHERE cust_ID =" &
    Request.Form("cust_ID")
'submit the UPDATE statement to the DBMS
Set objRecordset = objConn.Execute (queryString)
End Sub
&>
```

技巧 470 通过 Internet 处理 SQL 的事务处理过程

SQL 事务处理过程允许把多个 SQL 语句看作单个工作单元。根据关系数据库处理规则，要么 DBMS 成功地执行事务处理内的所有语句，要么就要取消由某些语句所完成的工作。换句话说，如果事务处理内的一条语句失败，DBMS 将使数据库内的数据看起来好像没有执行过什么语句一样。

如果有打开的事务处理，而且嵌入 Web 页面的脚本中止或是如果用户关闭了与 DBMS 的连接而没有执行 COMMIT 语句，则 DBMS 负责取消所完成的所有工作并将数据库表恢复到原始的未经修改的状态。用户既可明显地（通过调用 ADO Connection 对象的 Close 方法）关闭与 DBMS 的连接，也可隐含地（通过移动到另一 Web 页面或完全断开与 Internet 的连接）关闭与 DBMS 的连接。除此之外，长时期的不活动状态也将使 DBMS 关闭一个打开的连接，即使用户仍然处于与脚本用于打开与 DBMS 连接的同一 Web 页面上。

ADO Connection 对象有 3 个方法，可用来管理 SQL 的事务处理：

- **BeginTrans** 开始新事务处理。如果 DBMS 产品允许嵌套事务处理，可使用多个 BeginTrans 调用，而不必调用 CommitTrans 或 RollbackTrans 方法来关闭打开的事务处理。作为对照，如果 DBMS 不支持嵌套的事务处理，则当连接已经有了打开的事务处理，再调用此方法，则产生错误。
- **CommitTrans** 使最近一个 BeginTrans 方法调用以前由语句完成的工作永久化并结束（或关闭）当前的事务处理。如果 DBMS 产品支持嵌套的事务处理，每个 CommitTrans 方法调用关闭最内层的事务处理。这样一来，如果脚本已经作了 3 次 BeginTrans 方法调用（没有任何 RollbackTrans 或 CommitTrans 调用），第一个 CommitTrans 方法调用关闭并使第三个 BeginTrans 方法调用以来所完成的工作永久化。第二个 CommitTrans 方法调用使用第二个事务处理开始以来（且在现在已关闭的第三个事务处理之前）所完成的未提交的工作永久化。最后，第三个 CommitTrans 方法调用使第一个事务处理开始以来且在（现在关闭的）第二个事务处理之前所完成的工作永久化。
- **RollbackTrans** 取消所有自最后一条 BeginTrans 方法调用以来所发生的变化并结束当前的事务处理。如果 DBMS 产品支持嵌套的事务处理，每次 RollbackTrans 方法调用关闭最内层的事务处理，而让外层的事务处理仍然打开，同时让其内语句所完成的工作留在原地不变。

除非作出 BeginTrans 方法调用打开事务处理，DBMS 自动地提交（也就是使之永久化）由调用 Connect 或 Command 对象的 Execute 方法所提交的语句所完成的工作。因而，如果有下列语句流，没有工作被 RollbackTrans 方法调用所取消，因为语句是在打开的事务处理之外执行的（因而其工作已经提交）：

```
<%
  objConn.Execute "<SQL statement - MOD 1...>"
  objConn.Execute "<SQL statement - MOD 2...>"
  objConn.Execute "<SQL statement - MOD 3...>"
  objConn.RollbackTrans 'undoes nothing
%>
```

在调用 RollbackTrans 方法之后，3 条修改仍然有效，因为其工作已经被 DBMS 自动提交。

作为对照，如果有下列语句流，在事务处理仍然打开时，DBMS 并不自动提交（使之永久化）任何所完成的工作：

```
<%
  objConn.Execute "<SQL statement - MOD 1...>"
  objConn.BeginTrans
  objConn.Execute "<SQL statement - MOD 2...>"
  objConn.Execute "<SQL statement - MOD 3...>"
  objConn.RollbackTrans 'undoes MOD 2 & MOD 3
%>
```

RollBackTrans 方法调用取消了由 MOD 2 和 MOD 3 所完成的工作并关闭打开的事务处理。由 MOD 1 所完成的工作仍然留在原处，但是，由于 DBMS 自动提交了 MOD 1 的工作，因为它出现在打开的事务处理之外。

类似地，如果有下列语句流，CommitTrans 方法调用使 MOD 1 完成的工作永久化，而 RollbackTrans 方法调用将只取消由 MOD 2 和 MOD 3 所完成的工作：

```
<%
  objConn.BeginTrans
  objConn.Execute "<SQL statement - MOD 1...>"
  objConn.CommitTrans 'makes permanent MOD 1
  objConn.BeginTrans
  objConn.Execute "<SQL statement - MOD 2...>"
  objConn.Execute "<SQL statement - MOD 3...>"
  objConn.RollbackTrans 'undoes MOD 2 & MOD 3
%>
```

最后，当处理嵌套的事务处理时，如下面的事务处理，一条 CommitTrans 或 RollbackTrans 方法只影响当前的最内层事务处理内所完成的工作：

```
<%
  objConn.BeginTrans
  objConn.Execute "<SQL statement - MOD 1...>"
  objConn.Execute "<SQL statement - MOD 2...>"
  objConn.BeginTrans
  objConn.Execute "<SQL statement - MOD 3...>"
  objConn.BeginTrans
  objConn.Execute "<SQL statement - MOD 4...>"
  SQL_583-601.qxd 2/11/02 11:18 AM Page 1087
  objConn.CommitTrans 'makes permanent MOD 3
```

```
objConn.RollbackTrans 'undoes MOD 1 & MOD 2  
>
```

在本例中,第一条 RollbackTrans 方法调用取消了由 MOD 4 所完成的工作并关闭了最内层(第三级)事务处理。MOD 1、MOD 2 和 MOD 3 仍然不变,直到第一条 CommitTrans 方法调用使 MOD 3 完成的工作永久化并关闭最内层(第二级)事务处理。最后,第二条 RollbackTrans 方法调用取消了由 MOD 1 和 MOD 2 所完成的工作并关闭其余打开的事务处理。

技巧 471 创建与 MS-SQL Server 的虚拟连接

如果使用 MS-SQL Server DBMS,毫无疑问应该熟悉 SQL Query Analyzer,因为 SQL Query Analyzer 是 MS-SQL Server 提供的客户程序,使用它可登录到 DBMS 并处理数据库对象。可使用 Query Analyzer 提交标准的 SQL 语句以及只能在 MS-SQL Server 中使用的 Transact-SQL 命令以及内建的系统存储过程。

虽然功能强大而且对用户来说界面也是友好的,但 SQL Query Analyzer 仍然只是一种正好可与 MS-SQL Server 通信的应用程序。通过阅读本书中的各个技巧,读者已经学习了编写 Visual Basic 程序(技巧 298~技巧 329)和 Visual C++ 应用程序(技巧 280~技巧 297),这些程序都可向 DBMS 提交 SQL 语句和 Transact-SQL 命令,并从中提取数据。在技巧 354~技巧 365 中,读者学习了如何使用 ODBC 驱动程序、OLE DB 提供程序以及 ADO 对象,让嵌入 ASP 和 PHP Web 页面的脚本也与 MS-SQL Server DBMS 通信。

在本技巧中,读者将学习如何通过 Internet Information Server (IIS) 创建 MS-SQL Server 的虚拟连接。这些虚拟连接可用作让用户使用超文本传输协议(HTTP)直接与 DBMS 通信的管线。换句话说,在 MS-SQL Server 上建立了虚拟连接之后,可在 Web 浏览器的地址栏中键入 SQL 语句和 Transact-SQL 命令并将那些语句和命令直接发送给 MS-SQL Server,而不必通过 PHP 引擎或 ASP 脚本宿主。除了向 DBMS 发送命令和语句之外,可让 MS-SQL Server 将查询的结果集返回为 Web 浏览器可显示的 XML 文档。简短地说,MS-SQL Server 的虚拟连接允许用户使用 Web 浏览器(运行在 PC 上或其他有 Web 功能的装置)或其他能够发送和接受 HTTP 消息的应用程序来管理并处理 MS-SQL Server 数据库对象。

为了通过 IIS Web 服务器创建与 MS-SQL Server DBMS 的虚拟连接,可执行以下步骤:

1. 在 Web 服务器计算机的硬盘上为每个想要让 HTTP 协议访问的数据库创建一个文件夹。由于 Web 服务器必须有对这些文件夹的读取和写入权限,最好在 IIS 服务器的根目录内创建这些文件夹。例如,在根目录为 D:\inetpub\WWWRoot 的 IIS Web 服务器上,可为与 NORTHWIND 数据库的虚拟连接创建一个文件夹: D:\inetpub\WWWRoot\Nwind。

2. 在步骤 1 中创建的文件夹内,创建两个子文件夹: SCHEMA 和 TEMPLATE。例如,如果如步骤 1 所指明的创建 NWind 文件夹,可将 SCHEMA 和 TEMPLATE 文件夹分别创建为 D:\inetpub\WWWRoot\NWind\Schema 和 D:\inetpub\WWWRoot\NWind\Template。

3. 单击 Windows 的 Start 按钮,Windows 显示 Start 菜单,选择 Programs,指向 Microsoft SQL Server 程序组,然后单击 Configure SQL XML Support in IIS (配置 IIS 中的 SQL XML 支持)。Windows 将在类似于图 25.18 的窗口中启动 IIS Virtual Directory Management for SQL Server 程序。

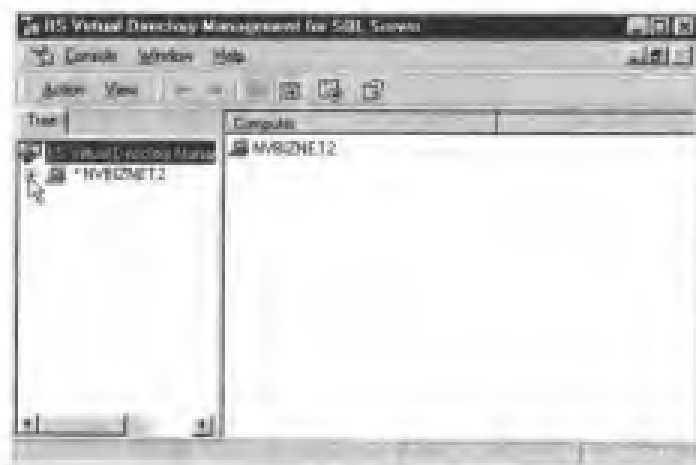


图 25.18 IIS Virtual Directory Manage for SQL Server 窗口

4. 单击其中有 Web 站点（想要通过此站点访问 MS-SQL Server）的 IIS Web 服务器左边的加号（+），配置程序将显示由该 Web 服务器管理的 Web 站点。

5. 单击在其中想要创建数据库虚拟目录的 Web 站点。对本例来说，单击 Default Web Site。

6. 选择 Action 菜单中的 New 选项并单击 Virtual Directory。配置程序将打开 New Virtual Directory Properties（新虚拟目录属性）对话框，如图 25.18 所示。

7. 在 General 选项卡的 Virtual Directory Name 域中键入虚拟目录的名称。当在 Web 浏览器的地址栏中指定将 SQL（或 Transact-SQL）语句发送到的数据库时要使用这个虚拟目录名，因而应该使用某种能够反映数据库名称的虚拟目录名。对本例来说，可在 Virtual Directory Name 域中键入 Northwind。

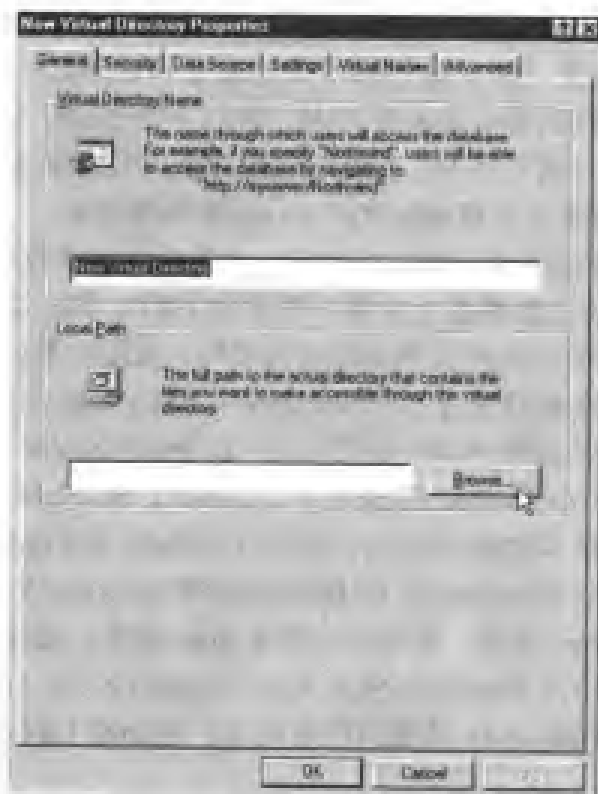


图 25.19 New Virtual Directory Properties 对话框的 General 选项卡

8. 在 Local Path 域内键入指向在步骤 1 中创建的虚拟连接的“根”文件夹的路径名。对本例来说, 可在 Local Path 域内键入 D:\InetPub\WWWRoot\NWind。

9. 在 Security 选项卡(如图 25.20 所示), 指定想让 MS-SQL Server 在允许 HTTP 访问数据库时的认证方法。如果在 Security 选项卡的 Credentials 域内键入用户名和密码, 要确保所指定的用户名只有对表的 SELECT 访问权, 除非想让该表对 Internet 上的所有人开放。请记住, 通过 Web 访问数据库的任何人都有此处所指定用户名的访问权限。为了提示用户输入用户名/密码, 既可选择 Use Windows Integrated Authentication (使用 Windows 集成的认证方法), 对有合法的 Windows NT/2000/XP 账户以及合法的 MS-SQL Server 账户允许访问, 也可选择 Use Basic Authentication (Clear Text) to SQL Server Account (对 SQL Server 账户使用基本认证[明文]方法), 对有合法的 MS-SQL Server 账户允许访问。对本例来说, 单击 Use Basic Authentication (Clear Text) to SQL Server Account 提示用户输入合法的 MS-SQL Server 用户名/密码对。

10. 在 Data Source (数据源) 选项卡上(如图 25.21 所示), 在 SQL Server 域中输入要有要设置 HTTP 访问数据库的 MS-SQL Server 名。或者单击该域右边的搜索按钮让配置程序搜索网络并在 Select Server (选择服务器) 对话框中显示可用的 MS-SQL Servers 清单, 然后单击 MS-SQL Server 名再单击 Select Server 对话框中的 OK 按钮。



图 25.20 New Virtual Directory Properties 对话框的 Security 选项卡



图 25.21 New Virtual Directory Properties 对话框的 Data Source 选项卡

11. 在 Database 域中, 输入设置 HTTP 访问的数据库名。对本例来说, 在该域内输入 Northwind。

12. 在 Settings (设置) 选项卡(如图 25.22 所示)中, 单击 Allow URL queries、Allow template queries、Allow XPath 和 Allow POST 复选框使之出现选择标记。

13. 在 Virtual Names (虚拟名称) 选项卡(如图 25.23 所示)中, 创建用户用于处理模板(在 TEMPLATE 子文件夹内)、架构(在 SCHEMA 子文件夹内)和数据库内的数据库对象时所使用的虚拟名称。单击 New 按钮。配置程序将显示 Virtual Name Configuration (虚拟名称配

置)对话框,如图 25.24 所示。

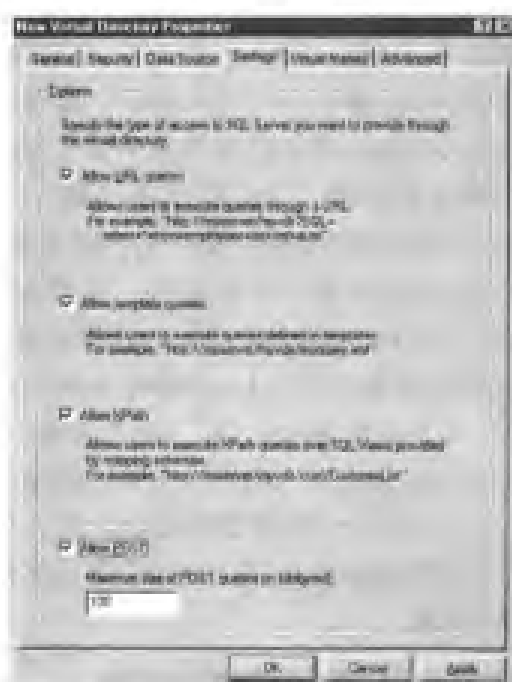


图 25.22 New Virtual Directory Properties 对话框的 Settings 选项卡



图 25.23 New Virtual Directory Properties 对话框的 Virtual Names 选项卡

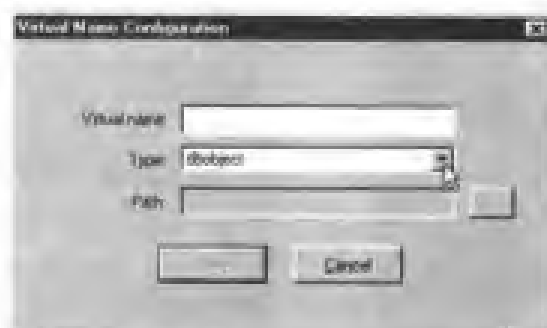


图 25.24 Virtual Name Configuration 对话框

14. 在 Virtual name (虚拟名称) 域中输入 schema。
15. 单击 Type 域右边的下拉按钮, 选择 Schema。
16. 在 Path (路径) 域中输入指向在步骤 2 中创建的 SCHEMA 子文件夹的路径。对本例来说, 在本域内输入 D:\InetPub\WWWRoot\NWind\Schema。然后单击 Save 按钮。配置程序将返回 Virtual Names 选项并将将在步骤 14 中输入的虚拟名称添加到 Virtual Names 选项卡内的 Defined virtual names (定义虚拟名称) 列表框中。
17. 在 Virtual Names 选项卡上单击 New 按钮。配置程序将再次显示 Virtual Name Configuration 对话框 (如图 25.24 所示)。
18. 在 Virtual name 域中输入 template。
19. 单击 Type (类型) 域右边的下拉列表按钮, 从选择列表中选择 template。
20. 在 Path (路径) 域中输入指向在步骤 2 中创建的 TEMPLATE 子文件夹的路径。对本

例来说, 在本域内输入 D:\inetpub\WWWroot\NWind\Template, 然后单击 Save 按钮。配置程序将返回 Virtual Names 选项并将在步骤 18 中输入的虚拟名称添加到 Virtual Names 选项卡内的 Defined virtual names (定义虚拟名称) 列表框中。

21. 在 Virtual Names 选项卡上单击 New 按钮。配置程序将再次显示 Virtual Name Configuration 对话框 (如图 25.24 所示)。

22. 在 Virtual name 域中输入 dbobject。

23. 单击 Type (类型) 域右边的下拉列表按钮, 从选择列表中选择 dbobject。然后单击 Save 按钮。配置程序将返回 Virtual Names 选项, 并将在步骤 21 中输入的虚拟名称添加到 Virtual Names 选项卡内的 Defined virtual names (定义虚拟名称) 列表框中。

24. 单击 New Virtual Directory Properties 对话框底部的 OK 按钮。

在完成步骤 24 之后, 配置程序将在步骤 5 中所选择的 IIS Web 服务器的 Web 站点上创建 Northwind 虚拟连接并返回 IIS Virtual Directory Management for SQL Server 窗口。

为了测试所创建的虚拟连接, 请启动 Web 浏览器 (以及拨号连接, 如果必要的话)。然后在浏览器的地址栏内使用如下句法输入一条简单的查询:

```
http://<Web Site Address>/<virtual connection>?sql=
<sql statement>+FOR+XML+AUTO&root=root
```

虽然此处显示的是两行, 但应在地址栏内作为一行键入。

例如, 如果在 Web 站点 www.NVBizNet2.com 上创建了 Northwind 虚拟连接, 可在浏览器的地址栏内键入查询如下, 然后按 Enter 键:

```
http://www.nvbiznet2.com/Northwind?sql=
SELECT+*+FROM+shippers+FOR+XML+AUTO&root=root
```

Web 浏览器将对 NORTHWIND 数据库内的 SHIPPERS 表执行以下查询:

```
SELECT * FROM shippers
```

请注意, 在浏览器的地址栏中键入时要用加号 (+) 代替上面语句中的空格。

DBMS 将让 Windows 提示用户输入合法的 MS-SQL Server 用户名/密码对, 然后将查询结果作为 XML 文档返回给 Web 浏览器, 如图 25.25 所示。



图 25.25 带有来自 SQL 查询结果的 XML 文档 (查询是使用 HTTP 向 MS-SQL Server 提交的)

技巧 472 使用 HTTP 执行 SQL 语句

在创建了与 MS-SQL Server 的虚拟连接之后 (通过执行技巧 471 “创建与 MS-SQL Server 的虚拟连接” 中的步骤), 就可使用此虚拟连接向 DBMS 发送想要发送的 SQL 语句或其他命

令。虚拟连接的作用像一条管道，让 HTTP 信息流流入和流出 DBMS。如果消息包括 SQL 查询，DBMS 将执行 SELECT 语句并向 Web 浏览器（或其他应用程序）以 XML 文档形式返回结果集。图 25.25 显示了 Internet Explorer 是如何在 XML 文档中显示返回的查询结果的。在本技巧中，读者将学习如何使用 HTTP 执行 SQL 和 Transact-SQL 语句。接着，在下一技巧中，读者将学习如何使 DBMS 与 XML 文档一起包括 XSL 样式表（XSL 样式表指示 Web 浏览器对它在 XML 文档中发现的 XML 定义的条目做什么，这样一来 Web 浏览器就可在 HTML 表内显示查询结果，而不是以原始的 XML 代码来显示）。

能够使用虚拟连接向 MS-SQL Server 发送基于 HTTP 的查询是非常好的功能，因为这使得数据库内可用的数据可在世界的任何地方通过 Internet 在 Web 浏览器内显示出来。但是，虚拟连接的真正的强大功能在于以下事实：使用虚拟连接可发送任何想让 DBMS 执行的命令，包括修改数据库内数据、创建或改变数据库对象的结构、添加或是删除用户、执行存储过程等的 Transact-SQL 语句。

由于通过虚拟连接访问 DBMS 的用户可采取任何有权限的行动，因而在为虚拟连接设置安全架构时一定要小心。在前一技巧的步骤 9 中，不要提供能够使用虚拟连接登录到 DBMS 的用户名/密码——除非该用户名只对特定的数据库对象有 SELECT 权限，其中的数据想要让 Internet 上的所有人都可使用。虚拟连接将把它所接收到的 HTTP 消息传递给 DBMS。DBMS 再执行那些虚拟连接的用户具有必要权限的语句。

为了把语句通过 Web 浏览器发往 DBMS，可在浏览器的地址栏内使用以下句法键入语句：

```
http://<IIS Web Site Address>/<name of virtual connection>
?sql=<statement string>+FOR+XML+AUTO&root=root
```

这样，为了通过 NVBizNet2.com 站点上的 Northwind 虚拟连接发送下面的 select 语句：

```
SELECT CompanyName, ContactName, City, Country
FROM suppliers
WHERE Country <> 'USA'
ORDER BY City, Country
```

可在 Web 浏览器的地址栏内键入以下的 URL：

```
http://www.NVBizNet2.com/Northwind/?sql=SELECT+CompanyName,
+ContactName,+City,+Country+FROM+suppliers+WHERE+Country+<>
+'USA'+ORDER+BY+City,+Country+FOR+XML+AUTO&root=root
```

请注意，一定要用加号（+）代替 SQL 语句或 transact-SQL 命令内的词汇之间的空格。

为了让 DBMS 执行存储过程，可使用类似的句法，在 DBMS 要执行的语句字符串中包括存储过程的参数值（如果有的话）：

```
http://<IIS Web Site Address>/<name of virtual connection>
?sql={EXEC|EXECUTE}<stored procedure name>
    [+@<parameter>=<'value'>]
    [...,+@<last parameter>=<'last value'>']&root=root
```

注意 URL 中没有 +FOR+XML+AUTO（用于作为 XML 文档内的 XML 标记插入列名）。如果存储过程返回结果集，必须在存储过程内的查询的尾部追加 FOR XML AUTO 字样（一会就会看到）。如果存储过程对数据库执行某些操作而不返回行集和列值，可完全忽略 FOR XML AUTO。

例如，为了让 DBMS 执行如下定义的存储过程 TenMostExpensiveProducts：


```
CREATE PROCEDURE TenMostExpensiveProducts AS
SET ROWCOUNT 10
SELECT ProductName, UnitPrice
FROM products
ORDER BY UnitPrice DESC
FOR XML AUTO
```

可向浏览器的地址栏内键入下面的 URL:

```
http://www.NVBizNet2.com/Northwind/?sql=
EXECUTE+TenMostExpensiveProducts&root=root
```

当然要使用自己的 Web 站点地址和虚拟连接名代替 www.NVBizNet2.com 和 Northwind。

如果必须将参数值传递给存储过程,可在键入地址栏内的 URL 中通过@<parameter name>=<value>一类的名称来指定每个参数。例如,可向 Web 浏览器的地址栏内键入以下 URL:

```
http://www.nvbiznet2.com/Northwind/?sql=EXECUTE+SalesByYear
+@Beginning_Date='06/01/1996',+@Ending_Date='05/31/1997'
&root=root
```

从而让 DBMS 执行如下定义的存储过程:

```
CREATE PROCEDURE SalesByYear
    @Beginning_Date DateTime, @Ending_Date DateTime AS
SELECT Orders.ShippedDate, Orders.OrderID,
    'Order Subtotals'.Subtotal,
    DATENAME(yy,ShippedDate) AS Year
FROM Orders INNER JOIN 'Order Subtotals'
    ON Orders.OrderID = 'Order Subtotals'.OrderID
WHERE Orders.ShippedDate
    BETWEEN @Beginning_Date AND @Ending_Date
ORDER BY Year
FOR XML AUTO
```

技巧 473 使用 XML 架构利用 HTTP 提交查询并使用 XSL 样式表来格式化查询结果

在前一技巧中,读者学习了如何使用 HTTP 向 MS-SQL Server 提交 SQL 语句和 Transact-SQL 命令。一般来说,在创建了 Internet Information Server(IIS)Web 服务器与 MS-SQL Server 上的服务器之间的虚拟连接(此内容已在技巧 471“创建与 MS-SQL Server 的虚拟连接”中学习过)之后,就可使用 HTTP 提交任何想让 DBMS 执行的语句字符串。只要在 Web 浏览器的地址栏内与虚拟连接的 Web 地址(也就是 URL)一起键入想要执行的语句。使得 DBMS 虚拟连接如此功能强大的是,可在任何地点由 Internet 通过虚拟连接可以查询、更新并管理数据库对象。

不幸的是,如果没有 XSL 样式表,大多数 Web 浏览器不知道对包括 DBMS 返回的查询结果集的 XML 文档内所描述的条目做什么。例如,Internet Explorer(IE)将下面的基于 HTTP 的查询结果的 XML 文档显示为如图 25.26 所示的样子。

```
http://www.NVBizNet2.com/Northwind/?sql=SELECT+employeeID,+
FirstName,+LastName,+Title,+Photo+FROM+employees+
FOR+XML+AUTO&root=root
```

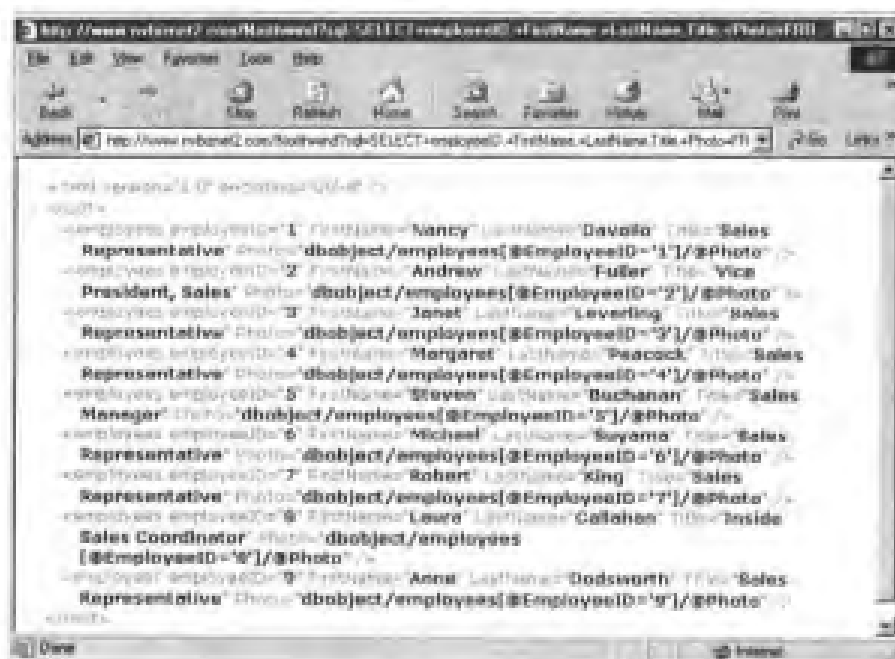


图 25.26 由 IE 显示的没有 XSL 样式表的 XML 文档

为了在 HTML 表内将查询结果显示为如图 25.27 中的样子,可创建一个如下的 XSL 样式表:

```
<?xml version='1.0' encoding='UTF-8'?>
<xsl:stylesheet xmlns:xsl='http://www.w3.org/TR/WD-xsl' >
  <xsl:template match = '*'>
    <xsl:apply-templates />
  </xsl:template>
  <xsl:template match = 'employees'>
    <TR>
      <TD><xsl:value-of select = '@EmployeeID' /></TD>
      <TD><xsl:value-of select = '@FirstName' /></TD>
      <TD><xsl:value-of select = '@LastName' /></TD>
      <TD><xsl:value-of select = '@Title' /></TD>
      <TD><B> <IMG><xsl:attribute name='src'>
        <xsl:value-of select = '@Photo' />
      </xsl:attribute>
      </IMG>
      </B></TD>
    </TR>
  </xsl:template>
  <xsl:template match = '/'>
    <HTML>
      <HEAD>
        <STYLE>th {background-color : lightblue }
        table {background-color : lightyellow}
        </STYLE>
        <BASE href='http://www.NVBizNet2.com/Northwind/'>
        </BASE>
      </HEAD>
      <BODY>
        <TABLE border='1' style='width:680;'>
```

```

<TR><TH colspan='5'>Employee Information
</TH></TR>
<TR><TH width='30'>ID</TH>
<TH width='100'>First Name</TH>
<TH width='100'>Last Name</TH>
<TH width='200'>Job Title</TH>
<TH width='250'>Photo</TH></TR>
<xsl:apply-templates select = 'root' />
</TABLE>
</BODY>
</HTML>
</xsl:template>
</xsl:stylesheet>

```

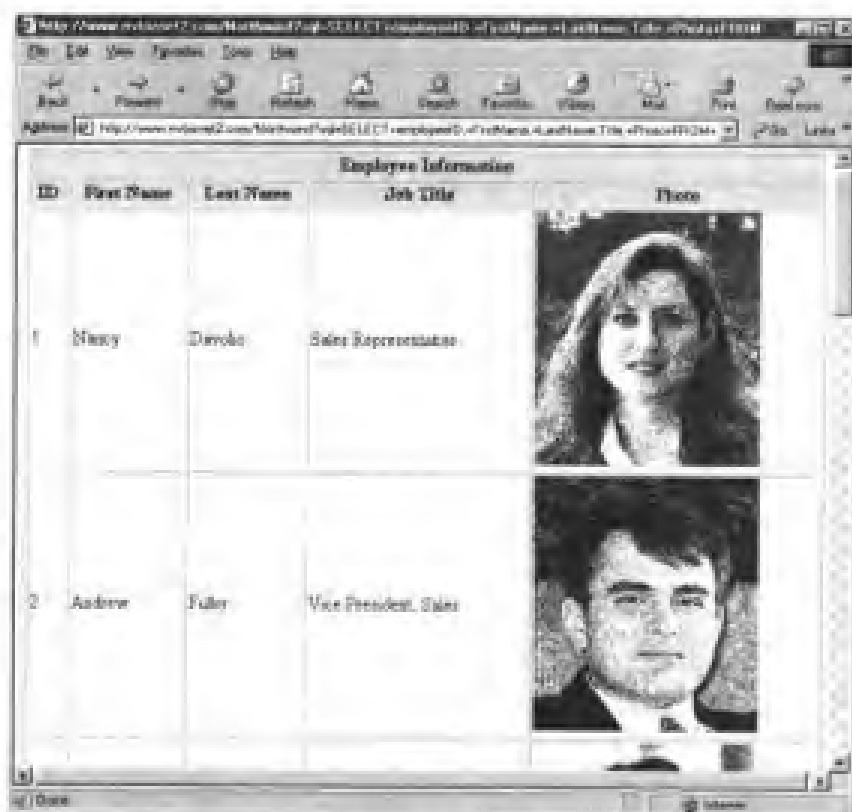


图 25.27 由 IE 所显示的有 XSL 样式表时的 XML 文档

虽然 XML 只是描述是什么组成了查询结果行之类的条目，而 HTML 标记告诉 Web 浏览器，如文本、图形图像、视频剪辑、动画和声音数据之类的 Web 页面对象应如何出现在屏幕上。换句话说，XML 描述了查询结果的单个项目，而 HTML 描述了 Web 浏览器如何显示各个项目。XSL 样式表告诉 Web 浏览器如何将 XML 文档内的条目转化为定义 Web 页面的 HTML 代码，从而可（以比普通文本更为优化的格式）显示这些条目。

例如，在图 25.27 之前的 XSL 样式表中，查询结果的每行都由 XML 文档中的下述代码所描述：

```

<employees
  employeeID='1' FirstName='Nancy'
  LastName='Davolio' Title='Sales Representative'

```

Photo="dbobject/employees[@EmployeeID='1']/@Photo" />

翻译成 HTML 为如下形式:

```
<TR>
  <TD>1</TD><TD>Nancy</TD>
  <TD>Davolio</TD><TD>Sales Representative</TD>
  <TD><B>
    <IMG src="dbobject/employees[@EmployeeID='1']/@Photo" />
    </B></TD>
</TR>
```

为达此目的, Web 浏览器将来自 XSL 样式表中的下面规则应用在本例中:

```
<xsl:template match = 'employees'>
  <TR>
    <TD><xsl:value-of select = '@employeeID' /></TD>
    <TD><xsl:value-of select = '@FirstName' /></TD>
    <TD><xsl:value-of select = '@LastName' /></TD>
    <TD><xsl:value-of select = '@Title' /></TD>
    <TD><B> <IMG><xsl:attribute name='src'>
      <xsl:value-of select = '@Photo' />
    </xsl:attribute>
    </IMG>
    </B></TD>
  </TR>
```

正如本技巧前面提到的, XSL 模板告诉 Web 浏览器用什么 HTML 标记和文本来代替 XML 文档内的 XML 条目引用。在本例中, XSL 样式表中的雇员模板告诉 Web 浏览器代替 XML 文档内每个雇员条目的 HTML 代码。这种由 XSL 样式表所提供的 HTML 代替使 Web 浏览器在 HTML 表内的行 (<tr></tr> 标记之间) 的单元格 (<td></td> 标记之间) 中显示每个雇员条目部分中的值 (employeeID、FirstName、LastName、Title、Photo)。在应用了 XSL 模板, 使所有必需的转换实现 (从 XML 条目描述转换为几组 HTML 标记、属性和文本) 之后, Web 浏览器将 XML 文档内返回的查询结果显示为 Web 页面内容, 如图 25.27 所示。

为了创建 XSL 样式表, 从而告诉 Web 浏览器如何显示来自提交给 DBMS 的基于 HTTP 的查询结果, 只要将本例中的 XSL 文件内显示的模板改变为与查询的 SELECT 子句中的列相匹配。一定要把在 XSL 开始模板描述行 (在本例中是 <xsl:template match = 'employees'>) 中所引用的引号括起的条目名 (employees) 改变为与模板应用于的 XML 条目相匹配。

例如, 为了处理由查询返回的 XML suppliers 描述:

```
SELECT CompanyName, ContactName, Phone
FROM suppliers
```

则应将前面示例 XSL 样式表中的模板描述改变为:

```
<xsl:template match = 'suppliers'>
  <TR>
    <TD><xsl:value-of select = '@CompanyName' /></TD>
    <TD><xsl:value-of select = '@ContactName' /></TD>
    <TD><xsl:value-of select = '@Phone' /></TD>
  </TR>
```

除此之外, 还应该将 XSL 文件靠近底部的 HTML 表描述 (在<table></table>标记内定义的) 改变为:

```
<TABLE border='1' style='width:230;'>
  <TR><TH colspan='5'>Supplier Information
    </TH></TR>
  <TR><TH width='30'>Company Name</TH>
    <TH width='100'>Contact Name</TH>
    <TH width='100'>Phone Number</TH>
  <xsl:apply-templates select = 'root' />
</TABLE>
```

为了让 Web 浏览器将 XSL 样式表文件内的模板应用于 XML 文档, 可在虚拟连接的根文件夹中创建 XSL 文件。接着, 在用来向 DBMS 提交查询的 URL 内引用该样式表。例如, 假设在文件 EmpNamePhoto.xml 中保存着想要应用于下面的基于 HTTP 的查询的 XSL 样式表:

```
http://www.NVBizNet2.com/Northwind/?sql=SELECT+employeeID,+
FirstName,+LastName,+Title,+Photo+FROM+employees+
FOR+XML+AUTO&root=root
```

通过在 URL 内的&root=root 前面插入&xsl=<xsl filename>引用, 可告诉 Web 浏览器在显示查询结果时应用 XSL 样式表 (EmpNamePhoto.xml), 如下所示:

```
http://www.NVBizNet2.com/Northwind/?sql=SELECT+employeeID,+
FirstName,+LastName,+Title,+Photo+FROM+employees+
FOR+XML+AUTO&xsl=EmpNamePhoto.xml&root=root
```

既然读者已经了解了如何解决在提交基于 HTTP 的查询时所遇到的显示问题, 现在要解决的惟一一件事是, 每次想要向 DBMS 提交查询 (即使是简单的查询) 时键入工作量的问题。这个问题的解决方法是把查询保存为 TEMPLATE 子文件夹 (在创建虚拟连接时所创建的) 中的一个 XML 模板文件。例如, 可将查询, 包括 XSL 样式表文件规格都保存在如下所示的 XML 模板文件内:

```
<?xml version = 1.0' encoding='UTF-8'?>
<root xmlns:sql='urn:schemas-microsoft-com:xml-sql'
  sql:xsl='../EmpNamePhoto.xml'>
  <sql:query >
    SELECT employeeID, FirstName, LastName, Title, Photo
    FROM employees FOR XML AUTO
  </sql:query>
</root>
```

假设在 TEMPLATE 子文件夹中的 EmpQuery.xml 内保存 XML 模板, 对于前面的基于 HTTP 的查询可键入 URL 如下:

```
http://www.NVBizNet2.com/Northwind/template/EmpQuery.xml
?contenttype=text/html
```

当然, 语句字符串越长, 则通过引用 XML 模板文件 (而不是键入查询本身) 所节省的键入工作量越大。

注意: 在本技巧中, 使用位于 www.NVBizNet2.com Web 站点上的名为 Northwind 的虚拟连接。当编写自己的基于 HTTP 的 SQL 语句时, 应该用自己定义虚拟连接的 Web 站点名来代替 www.NVBizNet2.com, 使用自己的虚拟连接名来代替 Northwind。还请注意, Web 站点和虚拟连接名出现在 XSL 样式表文件内的<base></base>标记之间。为了让数据库对象 (dobject) 引用正确

工作，必须代替本技巧中图 25.26 之后的示例 XSL 样式表中的 Web 地址和虚拟连接名。

技巧 474 显示保存在 SQL 表内的图像数据

为了显示保存在表内的图像数据（也就是图形图像、动画、视频剪辑或是其他二进制数据），Web 服务器端的脚本必须首先将图像数据写入磁盘文件。保存之后，嵌入图像出现其上的 PHP 或 ASP Web 页面的脚本必须将用于显示图像的标记内的 src 属性设置为脚本保存图像数据的文件的路径。在查看 ASP Web 页面用来提取并显示图像数据的 VBScript 之前，让我们先看一下如果使用的恰好都是 Microsoft 产品，这个过程是多么容易。

在 Web 页面显示 SQL 表内保存的图像涉及 3 个应用程序——SQL DBMS、Web 服务器和 Web 浏览器。如果使用 Microsoft 的 MS-SQL Server DBMS、Internet Information Server (IIS) Web 服务器和 Internet Explorer (IE) Web 浏览器，可在 Web 页面的 HTML 内插入如下的标记而显示保存在 SQL 表内的图像数据：

```
<img src=
"http://www.NVBizNet2.com/Northwind/dbobject/employees
[@employeeID='1']/@photo" >
```

标记的 src 属性指示 IE Web 浏览器向 MS-SQL Server 发送基于 HTTP 的查询。在本例中，Web 浏览器请求 DBMS 发送 EMPLOYEES 表内的 EMPLOYEEID 列值为 1 的行中的 PHOTO 列中的内容。当然，IE 仅当 NVBizNet2.com 站点上有名为 NORTHWIND 的与 DBMS 的虚拟连接，而且虚拟数据库对象 DBOBJECT 已经定义时，才能提取并显示图像数据（在技巧 471 “创建与 MS-SQL Server 的虚拟连接”中读者学习了如何建立提供对 MS-SQL Server 数据库对象的 HTTP 访问手段的虚拟连接）。

在技巧 472 “使用 HTTP 执行 SQL 语句”中，读者学习了如何在 Web 浏览器的地址栏内键入 URL 来提交 SQL 查询。使用那里学习的知识，就可显示保存在数据库表内的图像数据，只要在 Web 浏览器的地址栏内键入如下 URL：

```
http://www.NVBizNet2.com/Northwind/dbobject/Employees
[@EmployeeID='1']/@photo
```

再一次，为了让 IE 显示保存在 EMPLOYEES 表的 PHOTO 列中的图像数据，IIS Web 服务器上的 NVBizNet2.com 站点必须有与 MS-SQL Server 中包括 EMPLOYEES 表的数据库的名为 NORTHWIND 的虚拟连接。

最后，在技巧 473 “使用 XML 架构利用 HTTP 提交查询并使用 XSL 样式表来格式化查询结果”中，读者学习了如何在 XSL 样式表内编写如下标记定义，以便在 Web 页面上显示多幅图像：

```
<TD><IMG><xsl:attribute name='src'>
    <xsl:value-of select='@Photo'>/>
</xsl:attribute>
</IMG>
</TD>
```

遗憾的是，前面的 3 个在 Web 页面上显示图像数据的技术要求都使用 Microsoft 产品。为了使图像数据为各种 Web 浏览器和 DBMS 平台所用，可使用下面的嵌入 ASP Web 页面的 ADO 和 VBScript：

```

<%@ Language=VBScript %>
<% option explicit %>
<!-- #include file="adovbs.inc" -->
<%
'*****
'*** OPEN DSN-Less Connection ***
'*****
Sub open_OLEDB_connection (byVal ServerName, DbName,
                           Username, Password,
                           byRef objConn)
Dim connectString
  connectString = "PROVIDER=SQLOLEDB;DATA SOURCE=" & _
    ServerName & ";UID=" & UserName & ";PWD=" & _
    Password & ";DATABASE=" & DbName
  With objConn
    .ConnectionString = connectString
    .open
  End With
End Sub
'*****
'*** Variable Declarations ***
'*****
Dim objDiskAccess
Dim objConn
Dim objRecordset
Dim ADO_field_header
Dim block_size
'connection properties & query string
Dim Username
Dim Password
Dim ServerName
Dim DBName
Dim query_string
'image file processing variables
Dim block_count
Dim image_chunk
Dim image_file_extension
Dim image_file_size
Dim offset
Dim remainder
Dim temp_image_filename
Dim temp_image_pathname
Dim temp_image_physical_folder
Dim temp_image_virtual_folder
Dim html_image_tag
'*****
'*** Setup Constants ***
'*****
ADO_field_header = 78
block_size = 256

```

```

'Determine where you want the image stored on disk
'***Change these to match where you want the image stored.
temp_image_filename = "Image"
temp_image_physical_folder = _
    "D:\Inetpub\wwwroot\NWind\Temp\"
temp_image_virtual_folder = "/NWind/Temp/"
image_file_extension = ".bmp"
'ADO connection string properties
'*** Change these to match your DB access needs.
UserName = "username"
Password = "password"
ServerName = "NVBizNet2"
DBName = "Northwind"
'***Change the query string to retrieve the image you want
'***To retrieve from the DBMS
query_string = _
    "select Photo from Employees where EmployeeID='1'"
'*****
'*** Main Routine ***
'*****
on error resume next
'***You can download a copy of "FileAccessor.dll" from
' the book's companion Web page at
' www.PremierPressBooks.com/
Set objDiskAccess = _
    CreateObject("FileAccessor.FileWriter")
'Formulate the pathname for the image file the script will
' create on disk.
>Delete the previously written file of the same name (if it
' exists).
'Then open the disk file into which the script will write
' the contents of the image column from the table.
'***You need to change this "delete action" to match your
' image file retention requirements.
temp_image_pathname = _
    temp_image_physical_folder & temp_image_filename & _
    image_file_extension
objDiskAccess.RemoveFile temp_image_pathname
objDiskAccess.OpenFile temp_image_pathname
'Open a connection to the DBMS and call the method that
' executes the query.
Set objConn = Server.CreateObject("ADODB.Connection")
Open_OLEDB_Connection _
    ServerName, DbName, Username, Password, objConn
Set objRecordset = objConn.Execute (query_string)
'Compute the image file's size (in bytes) by subtracting
' the bytes within the ADO field header
'Then, after discarding the header bytes stored at the
' start of the image field's column within the ADO
' Recordset object, Compute the number of "block size"

```



```

' blocks there are within the image field.
image_file_size = _
    objRecordset.fields("Photo").ActualSize - _
    ADO_field_header
image_chunk = _
    objRecordset.fields("Photo").GetChunk(ADO_field_header)
block_count = image_file_size \ block_size
'To make the last write ouptput a full "block size"
' buffer, divide the block size into the image's total
' size and then retrieve and write to the disk file any
' "left over" bytes so that when looping the read you
' always have exactly some number of "block size" (and no
' extra) bytes to read and write
remainder = image_file_size Mod block_size
If Remainder > 0 Then
    image_chunk = _
        objRecordset.fields("photo").GetChunk(remainder)
    objDiskAccess.WriteToFile image_chunk
End If
'Work through the image field "block size" bytes at a time
' and append each block of bytes onto the disk file.
offset = remainder
Do While Offset < image_file_size
    image_chunk = _
        objRecordset.fields("photo").GetChunk(block_size)
    objDiskAccess.WriteToFile image_chunk
    offset = offset + block_size
Loop
html_image_tag = _
    "<img src='" & temp_image_virtual_folder & _
        temp_image_filename & image_file_extension & "'/>"
'Close the disk file, recordset and the DBMS connection.
objDiskAccess.CloseFile
objRecordset.Close
objConn.Close
%>
<html>
<head>
    <title>Display Image Data from Table Column</title>
</head>
<body>
    <center>
        <h1>"Photo" Image Column Contents</h1>
        <table border='2'>
            <tr><td><%=html_image_tag%></td></tr>
        </table>
    </center>
</body>
</html>

```

请注意，本例中的脚本从 NORTHWIND 数据库内的 EMPLOYEES 表的一行中提取 PHOTO

列的内容。但是，可把此程序定制为提取并显示任何图像列的内容。

除了任何 ASP 脚本引擎都可处理的 VBScript 之外，在本例的脚本中使用了名为 FileAccessor.DLL 的 ActiveX 控件中的磁盘访问方法。例如，可把 FileAccessor.DLL 保存在 Web 站点的根文件夹 D:\InetPub\wwwroot\中。不管将 ActiveX 控件保存于何处，一定要确保记下其位置，因为使用 REGSVR32.EXE 注册控件以后，ASP 脚本才能使用该控件的方法。例如，如果将控件保存在文件夹 D:\InetPub\wwwroot 内，可使用以下命令在 Web 服务器上注册 ActiveX 控件：

```
REGSVR32 D:\InetPub\WWWRoot\FileAccessor.DLL
```

如果已经有了 VB 应用程序，则可使用它来代替 FileAccessor.DLL。要理解的重要事情是，可使用 ADO 从数据库表中将图像数据提取到 VBScript 脚本中。接着必须使用外部程序（如 FileAccessor.DLL）将 ADO Recordset 对象中的数据写入硬盘文件中。正如在本技巧开头所提到的，必须将图像数据写入物理磁盘文件中，然后才能在 Web 页面上显示图像。

列的内容。但是，可把此程序定制为提取并显示任何图像列的内容。

除了任何 ASP 脚本引擎都可处理的 VBScript 之外，在本例的脚本中使用了名为 FileAccessor.DLL 的 ActiveX 控件中的磁盘访问方法。例如，可把 FileAccessor.DLL 保存在 Web 站点的根文件夹 D:\InetPub\wwwroot\中。不管将 ActiveX 控件保存于何处，一定要确保记下其位置，因为使用 REGSVR32.EXE 注册控件以后，ASP 脚本才能使用该控件的方法。例如，如果将控件保存在文件夹 D:\InetPub\wwwroot 内，可使用以下命令在 Web 服务器上注册 ActiveX 控件：

```
REGSVR32 D:\InetPub\WWWRoot\FileAccessor.DLL
```

如果已经有了 VB 应用程序，则可使用它来代替 FileAccessor.DLL。要理解的重要事情是，可使用 ADO 从数据库表中将图像数据提取到 VBScript 脚本中。接着必须使用外部程序（如 FileAccessor.DLL）将 ADO Recordset 对象中的数据写入硬盘文件中。正如在本技巧开头所提到的，必须将图像数据写入物理磁盘文件中，然后才能在 Web 页面上显示图像。